An Introduction to Machine Learning and ML tool basics for Physicists

Thomas Fischbacher

Google Research
Brandschenkestrasse 110, 8002 Zürich, Switzerland

tfish@google.com

Abstract

These lecture notes contain of a set of Interactive Python (Google Colab) notebooks that were the basis for a one-week ML course at the Albert Einstein Institute in Potsdam, taught by the author in June 2022. They give a hands-on introduction to not only Machine Learning Fundamentals and underlying concepts and ideas (in some places presented in a form that aligns with concepts known to theoretical physicists that many machine learning practitioners would not be familiar with), but also explain bottom-up some of the design goals of modern machine learning frameworks, and how these can be harnessed to solve a broad range of numerical problems that readily arise in theoretical physics - even when not employing any "learning" per se. This point is illustrated with examples such as a basic general relativity ray tracer implemented on top of Google's TensorFlow. The notebooks have been typeset with some minor expansions, fixes, and modernizations. It is expected that they might be useful both for self-study and also as a source of material for educators who want to design similar courses.

Contents

1 Preface **1**

2 A Whirlwind Tour of Python **2**
2.1 Why? . . . 2
2.2 Objective of this Lecture . . . 2
2.3 Colab Notebooks . . . 2
2.4 Python - Some History . . . 5
2.4.1 Python vs. Lisp . . . 5

3 Understanding Python **7**
3.1 Python Syntax . . . 7
3.1.1 Mathematical term grammar . . . 7
3.1.2 Python's Grammar . . . 9
3.2 Python's "Data Model" . . . 11
3.2.1 Key Aspects . . . 11
3.2.2 Important Types/Classes . . . 12
3.2.3 Other types . . . 16
3.2.4 Exceptions . . . 17
3.3 Python's "Execution Model" . . . 17
3.3.1 How Python code gets executed . . . 19
3.4 Useful things to know . . . 23
3.4.1 N-index Arrays (in their various incarnations) . . . 23
3.4.2 "Python as a Query Language" . . . 25
3.4.3 General principles for good code readability and maintainability . . . 29

4 Important References **31**

5 Proposed Candidate Python Coding Exercises **31**
5.1 Addendum: Graph structure of Python in-memory objects . . . 32

6 Derivatives, Numerical Optimization, and Machine Learning **39**
6.1 Why we are looking into this . . . 39

7 Exploring Derivatives **41**
7.1 Question (Trick question?): What is the best step size for `derivative_v0`? . . . 41
7.1.1 Answer (. . . which will lead us to FM-AD and RM-AD.) . . . 42

8 Before we move on. . . a Python interlude. **43**

9 Reverse Mode AD, Step-by-Step **45**
9.1 Computational Complexity . . . 51
9.2 More complicated code structure . . . 52

10 Actually doing optimization **57**
10.1 Let us improve this by adding fast gradients. . . . 61
10.1.1 Hints for implementing grad_heron_area . . . 66
10.1.2 Testing it out . . . 66

11 A first Machine Learning Example **67**
11.1 Participant Exercises . . . 77

12 Discussion **85**
12.1 How things can go wrong . . . 86
12.1.1 Adversarial attacks . . . 86
12.1.2 Generative Models . . . 87

13 Appendices **87**
13.1 Complex Backpropagation . . . 87

14 Basic ML Concepts and Ideas **88**
14.1 Overview . . . 88
14.2 More about our MNIST classifiers . . . 88
14.2.1 The Loss Function . . . 89
14.2.2 The Logistic ("sigmoid") Function . . . 92
14.2.3 Multiclass classification . . . 94
14.2.4 More about the ("pedestrian"-)thermodynamics interpretation . . . 95

15 Some Concepts and Terminology **96**
15.1 A DL Classifier "from scratch" . . . 98
15.2 Summary . . . 107
15.3 A Closer Look . . . 107
15.4 Addendum: Thoughts on universal approximation theorems . . . 115

16 Machine Learning with TensorFlow **116**
16.1 TensorFlow for the ML Practitioner . . . 116
16.1.1 What TensorFlow did for us here . . . 125
16.1.2 What we have not seen yet . . . 126
16.1.3 Additional remarks . . . 126
16.1.4 Architecture Improvements . . . 126

17 Other major architectural ideas **132**
17.1 Embeddings . . . 132
17.2 The wider ML Landscape . . . 134

18 Using TensorFlow for Physics **135**
18.1 Accelerating computations with Graphics Hardware . . . 135
18.2 TensorFlow 2 and Python . . . 145
18.3 Some Physics . . . 150
18.4 Rendering a Black Hole with TensorFlow . . . 155

19 Advanced Topics (. . . tying up some ends. . .) **177**
19.1 Extending Keras . . . 177
19.2 TensorFlow and JAX . . . 182

20 Recap **183**
20.1 Addendum: Connecting Mathematica and TensorFlow . . . 184
20.1.1 Having Mathematica use an external HTTP server to access ML capabilities . 186

1 Preface

These lecture notes give an introduction to Machine Learning (henceforth, "ML") basics that can be conveniently covered in a one-week compact course for a target audience of participants with some background in physics. It was first taught in June 2022 (virtually) at the Albert Einstein Institute in Potsdam, as part of the International Max Planck Research School, and video recordings of the lectures are available on the AEI web page. This document mostly consists of the notebooks that were presented during the course, with some fixes, amendments, and extensions (and this origin explains the somewhat messed-up structure of the table-of-contents). The arXiv web page for these lecture notes also have these notebooks as standalone `.ipynb` interactive Python notebook documents, which interested readers can download and study - and in case they have a `@gmail.com` email account - hence a Google account - upload to their Google Drive (`http://drive.google.com/`) and then execute as Google Colab notebooks. This approach gives a convenient way to learn about Machine Learning without having to go through the motions of installing any software locally - since all code will run on a virtual machine back-end in the Google cloud that has the relevant ML libraries installed.

Given that there are many ML tutorials available nowadays, the question seem valid why to introduce yet another one. The two primary reasons behind this are that, first, there are some fundamental constructions in ML that are interesting to discuss but much easier to explain to participants with some background on differential equations, and second, that the big public attention focus on the current ML megatrend can lead to overlooking major gold veins in the mine. In particular, just as ML is benefitting a lot from the computer gaming industry having made powerful numerics hardware very affordable, physics can benefit a lot from not only ML itself, but also the ML having made high-dimensional numerical optimization much easier to handle than in the past. While it has been possible in principle for many years to write fast gradient code even for complicated calculations by hand (as the present author has e.g. demonstrated back in 2008 in https://arxiv.org/abs/0811.1915 - The many vacua of gauged extended supergravities [2]), pragmatically speaking, this was effectively out of reach for many practitioners in physics due to the high time commitment required to both master the technique and then write the actual code. A major thesis of this course is that with ML technology available, just about every physicist should be empowered to readily tackle many a non-malicious numerical optimization problem in 1000-dimensional parameter space - and the course material discusses in detail how to.

Apart from these primary reasons, there also are secondary reasons for presenting this material in the form chosen here. One of them obviously is to showcase the general usefulness of Jupyter notebooks (here in their incarnation as Google Colab documents). On the educational side, this course emphasizes the importance of forming good mental models for basic principles and ideas, empowering the participant to correctly and proficiently reason about behavior - to the largest extent possible. This empowerment especially is an explicit articulated course objective here since there is a very concerning trend in recent years for technical documentation to move in the opposite direction and merely provide do-this-to-use-it instructions; "the opposite of education is training".

One obvious obstacles to this important objective of always enabling the student to correctly reason about behavior is of course limited available time. Clearly, the course needs - for example - to contain a quick introduction to the Python programming language, but cannot possibly cover all the relevant detail in about a single day.

Whenever this problem arises, the course at least tries to provide references to the authoritative documentation, which sometimes may itself have shortcomings, but nevertheless generally provides

the best-available answers.

Another important obstacle to "enabling students to reason about behavior" is that we generally cannot reason that well about what is going on deep inside ML models - especially in comparison to algorithmic code. Overall, this is not at all surprising: once we consider employing ML to handle a problem, this typically means we already have conceded defeat on the algorithmic front - that is, given up on the idea that we could come up with some precise algorithmic definition that could be inspected and proven to satisfy the expected objectives via stringent analytical reasoning. Given that we are still discovering and learning to understand a lot about in particular Deep Learning, we might be able to explain some things we still struggle with these days better in the future - but in general, we should perhaps not be too surprised to find that trying to reason about the behavior of ML systems can run into similar problems as trying to reason about biological systems.

2 A Whirlwind Tour of Python

2.1 Why?

- Many ML frameworks in use today are Python-based.
- In general, tasks around "ML model definition" are usually done in Python.
- When a trained ML model gets deployed in a product, Python will in general not be involved.

2.2 Objective of this Lecture

- Give a reasonably solid overview over the skeletal structure of the Python programming language.
  - "These are the basic ideas and principles"
  - "Authoritative documentation is over there"
  - "Extension Modules/Libraries are documented here"
  - General principle: To the extent we can do this given constraints, we will always reference the authoritative sources of truth for information about Python (rather than merely presenting examples).
- Explain basic design decisions underlying the language.
- Gain some familiarity with one particular Python execution environment that is useful for experimenting - Google Colab Notebooks.

2.3 Colab Notebooks

This document is a Google colab notebook. Much of this course will be notebook-based, since such notebooks are a UI concept many physicists (and data analysts) are already familiar with.

What are we seeing here? The current file is a document on Google Drive cloud storage (linked to a @gmail.com or @google.com account) that is an "interactive Python notebook" - it can be downloaded as such, using the menu (and re-uploaded again). It can also be downloaded as a .py Python file. This loses some information about text cells and their formatting. Also, Colab notebooks have some limited (single-file) version tracking capabilities, to explore questions such as "what did I change over the past 24 hours in this file?"

A Colab Notebook can be connected to a "Colab Runtime". Basically, this is an (emulated) Linux machine in the cloud that can execute "Interactive Python" (IPython) commands. The IPython

interpreter will forward lines of code that start with an exclamation mark (!) to the shell - so we can indeed run shell code on the runtime Virtual Machine (as a root-user!).

[ ]:
```
!head -n 20 < /proc/cpuinfo
```

```
processor       : 0
vendor_id       : AuthenticAMD
cpu family      : 23
model           : 49
model name      : AMD EPYC 7B12
stepping        : 0
microcode       : 0xffffffff
cpu MHz         : 2249.998
cache size      : 512 KB
physical id     : 0
siblings        : 2
core id         : 0
cpu cores       : 1
apicid          : 0
initial apicid  : 0
fpu             : yes
fpu_exception   : yes
cpuid level     : 13
wp              : yes
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov
pat pse36 clflush mmx fxsr sse sse2 ht syscall nx mmxext fxsr_opt pdpe1gb rdtscp
lm constant_tsc rep_good nopl nonstop_tsc cpuid extd_apicid tsc_known_freq pni
pclmulqdq ssse3 fma cx16 sse4_1 sse4_2 movbe popcnt aes xsave avx f16c rdrand
hypervisor lahf_lm cmp_legacy cr8_legacy abm sse4a misalignsse 3dnowprefetch
osvw topoext ssbd ibrs ibpb stibp vmmcall fsgsbase tsc_adjust bmi1 avx2 smep
bmi2 rdseed adx smap clflushopt clwb sha_ni xsaveopt xsavec xgetbv1 clzero
xsaveerptr arat npt nrip_save umip rdpid
```

[ ]:
```
!echo "### Disk Usage ###"
!du -h .
!df -h
!echo "### Linux System RAM ###"
!head -n 10 < /proc/meminfo
# The UI also shows ram/disk information about the connected runtime system
# in the top right corner.
```

```
### Disk Usage ###
72K     ./.config/logs/2023.08.14
76K     ./.config/logs
8.0K    ./.config/configurations
120K    ./.config
55M     ./sample_data
55M     .
Filesystem      Size  Used Avail Use% Mounted on
```

```
overlay         108G   27G   82G  25% /
tmpfs            64M     0   64M   0% /dev
shm             5.8G     0  5.8G   0% /dev/shm
/dev/root       2.0G  1.1G  887M  55% /usr/sbin/docker-init
tmpfs           6.4G  360K  6.4G   1% /var/colab
/dev/sda1        44G   28G   16G  65% /etc/hosts
tmpfs           6.4G     0  6.4G   0% /proc/acpi
tmpfs           6.4G     0  6.4G   0% /proc/scsi
tmpfs           6.4G     0  6.4G   0% /sys/firmware
### Linux System RAM ###
MemTotal:       13294252 kB
MemFree:         8461152 kB
MemAvailable:   12071020 kB
Buffers:           98472 kB
Cached:          3672952 kB
SwapCached:            0 kB
Active:           910260 kB
Inactive:        3608388 kB
Active(anon):       1324 kB
Inactive(anon):   747552 kB
```

The underlying runtime is a (modified) Debian GNU/Linux system.

We can install extra packages if we want to.

Runtime systems are allocated when we start a colab and get de-allocated if they remain idle for too long.

There of course are some "terms and conditions" around what colabs can and can not be used for. These can be found at: https://colab.research.google.com/pro/terms/v1.

Many relevant packages are pre-installed here, but let us install some extra Debian packages and also some extra Python packages which we will need further down in this specific notebook.

[ ]:
```
!apt install clisp # A Debian package
!pip install lark # A Python package
# (Output below is from a 2nd run; these install commands are idempotent
#  apart from producing lengthy output when installing a package.)
```

```
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
clisp is already the newest version (1:2.49.20210628.gitde01f0f-2).
0 upgraded, 0 newly installed, 0 to remove and 16 not upgraded.
Requirement already satisfied: lark in /usr/local/lib/python3.10/dist-packages
(1.1.7)
```

2.4 Python - Some History

- Crude but useful model: There are two complementary approaches to "computing": machine-oriented vs. maths-oriented.

  - Machine-oriented school: Focus on Turing machines, Knuth's "MIX Assembly", etc.
  - Maths-oriented: "What primitives do we need to meaningfully talk about algorithms?"
    * "Term evaluation" and "function application" can take us very far.
    * (This needs a suitable notion of "function" ~ computation specification.)
    * "More recently" also of importance (~since 1970+): Concepts from category theory. Especially: those related to "algebraic data types" and "composition operations".
- Modern ML is closely related to AI.
  - Historically, much of AI-related work since the 1960s has been done in LISP.
  - LISP follows the "maths-oriented" approach to computation.
  - Looking at the core language (sans libraries), many LISP systems are still more powerful than Python. Python became popular due to "being more accessible".
  - LISP's blessing and curse is its simplicity:
    * One uniform approach to express structured data:
      "Structure is generally built out of pairs" (called 'cons cells').
    * Programs are also directly written down as syntax trees!
    * A minimal LISP-like system:
      · Key atoms: "symbols" ("name tags" that provide "identity") and numbers.
      · "Pairs", written as `(left . right)` with extra rule "if a dot precedes a '(', we need not write the dot and the parentheses around the expression to its right".
      · "Evaluation rules" for "evaluating an expression specified as a tree".
      · Some built-in function definitions, like the functions `abs`, `+`, etc.
      · Most important non-function elements: "if"-conditional and "define-a-function-that-evaluates-this-term".
    * Lisp code, like sheet music, "looks alien to the un-initiated".
      · Entry barrier!
      · Requires discipline!
      · (...but great experience once mastered, due to structural simplicity bringing enormous flexibility).
    * Two of the three currently most popular programming languages conceptually are effectively LISP systems "with some syntax sugar on top to lower the entry barrier": JavaScript and Python.

2.4.1 Python vs. Lisp

We will have to say more about Python-vs-Lisp when we discuss the TensorFlow2 evaluation model. Here, we only have time for a quick - but important - first glimpse.

[ ]:
```
# This Python notebook Code cell
# - defines a multi-line string that contains some LISP code.
# - writes that LISP code to a file.
# - executes a LISP interpreter on that code.
# - Also contains a Python function that closely corresponds to the
#   LISP function in this file, for syntax comparison.

lisp_code_1 = """
(defun print-multiplication-table (n)
  (loop for i from 1 to 10
   do (format t "~2d x ~2d = ~2d~%" n i (* n i))))
```

```
"""

with open('/tmp/mt.lisp', 'wt') as h_out:
  h_out.write(lisp_code_1)
print('=== LISP ===')
!clisp -q -x '(load "/tmp/mt.lisp")' -x '(print-multiplication-table 7)'

###

def print_multiplication_table(n):
  for i in range(1, 11):
    print('%2d x %2d = %2d' % (n, i, n * i))
print('=== Python ===')
print_multiplication_table(7)
```

```
=== LISP ===
;; Loading file /tmp/mt.lisp ...
;; Loaded file /tmp/mt.lisp
#P"/tmp/mt.lisp"
 7 x  1 =  7
 7 x  2 = 14
 7 x  3 = 21
 7 x  4 = 28
 7 x  5 = 35
 7 x  6 = 42
 7 x  7 = 49
 7 x  8 = 56
 7 x  9 = 63
 7 x 10 = 70
NIL
=== Python ===
 7 x  1 =  7
 7 x  2 = 14
 7 x  3 = 21
 7 x  4 = 28
 7 x  5 = 35
 7 x  6 = 42
 7 x  7 = 49
 7 x  8 = 56
 7 x  9 = 63
 7 x 10 = 70
```

The similarities between Python's "def print_multiplication_table" and the LISP "defun print-multiplication-table" are quite evident.

However, the LISP code is merely a tree of symbols (and numbers and a string) that could equivalently be written as:

```
(defun . (print-multiplication-table . ((n . ()) . {...})
```

Here, indentation is purely arbitrary, and we used a single simple(!) rule to elide "spurious" parentheses.

For the Python code however, indentation carries meaning (like in Haskell). While LISP code generally is described as "having many parentheses", every LISP hacker reads and writes code by-indentation, and Python merely makes indentation carry actual meaning.

Participant Exercise: How many parentheses do we count in the Python and LISP definitions of the multiplication-table function?

3 Understanding Python

As for just about every programming language, understanding its structure and interpretation requires a solid understanding of three core aspects:

- The Syntax
- The "Data Model"
- The "Execution Model"

3.1 Python Syntax

While in LISP, one directly specifies an arithmetic expression as a tree, such as `(+ 2 (* 3 4))` ("the sum of 2 and (the product of 3 and 4)"), Python uses a dedicated software component that translates source code to an internal "syntax tree representation". This is the job of a "Parser".

Generally speaking, almost all programming languages use a "parser" to process source code, but such parsers are basically never written by hand. Rather, there are programs that take as input a grammar-specification (itself "a kind of program") and produce the code for a parser that processes input following this grammar.

3.1.1 Mathematical term grammar

Common mathematical notation also has a grammar - and the grammar of most programming languages mostly is an extension of that grammar. This term-grammar is generally introduced/discussed in 5th/6th grade mathematics, with a typical exercise of the form:

```
Translate the term `5+(7+9)/2` into structured form.
```

The expected answer being:

```
* This term is a sum.
  * The 1st summand is the number 5
  * The 2nd summand is a quotient.
    * The numerator is a parenthesized term.
      * This term is a sum.
        * The 1st summand is the number 7.
        * The 2nd summand is the number 9.
    * The denominator is the number 2.
```

This "structured form" is, of course, a tree (... which we could write in LISP tree notation). Let us actually write a parser for such a simplified term grammar, using a grammar and a parser-generator (the one provided by the "lark" Python module, which we installed and imported earlier).

[ ]:
```
import lark

# EBNF (Extended Backus-Naur Form) grammar, see:
# https://en.wikipedia.org/wiki/Extended_Backus%E2%80%93Naur_form
# https://lark-parser.readthedocs.io/_/downloads/en/latest/pdf
grammar = """
?start: sum

?sum: product
    | sum "+" product  -> add
    | sum "-" product  -> sub

?product: factor
    | product "*" factor -> mul
    | product "/" factor -> div

?factor: NUMBER -> num
    | "(" sum ")"

NUMBER: /[+-]?[0-9]+/
"""

parser = lark.Lark(grammar)

print(parser.parse("7"))
# Prints:
# Tree('num', [Token('NUMBER', '7')])

print('======')

print(parser.parse("1+2"))
# Prints:
# Tree('add', [Tree('num', [Token('NUMBER', '1')]), Tree('num', [Token('NUMBER',␣
 ↪'2')])])

# There also is pretty-printing for parsed trees:
print('(pretty-printed form)')
print(parser.parse("1+2").pretty())

def show_pretty(term):
  return '=== {} ===\n{}'.format(term, parser.parse(term).pretty())

print('======')

print(show_pretty("2+3*(4+5)"))
# Further examples:
# print(show_pretty("7*8*9/10"))
```

```
# print(show_pretty("2*3-4*5*6-7*8"))
# print(show_pretty("1+1/(1+1/(1+1/(1+1)))"))
```

```
Tree('num', [Token('NUMBER', '7')])
======
Tree('add', [Tree('num', [Token('NUMBER', '1')]), Tree('num', [Token('NUMBER',
'2')])])
(pretty-printed form)
add
  num   1
  num   2

======
=== 2+3*(4+5) ===
add
  num   2
  mul
    num 3
    add
      num       4
      num       5
```

3.1.2 Python's Grammar

Python's (and most other programming languages') grammar can be seen as a massive expansion of the commonly used grammar for mathematical terms. Here and in languages such as C, Java, etc that can be (mostly) parsed with a grammar, the typical extensions are:

- We have many more binary operators beyond the arithmetic ones, `+, -, *, /, {** or ^}`. Typically, there also is `<<, ~, &, %`, plus quite a few more.
- There are both "expressions" (which evaluate to a value, so "generalized mathematical terms"), and also "statements" which "describe instructions", and then there also is a notion of a "block of statements" - which are executed one-after-another.
- In some languages (including Python), "expressions" (which have a value) are a special kind of statements.

The Python language's full grammar specification can be found at: https://docs.python.org/3/reference/grammar.html. This then in particular settles questions such as what the specific role of a pair of round parentheses is in a piece of code, what operators there are, how operator precedence works, and what the keywords in the language are. Naturally, the grammar has some hierarchical structure, there is a sub-grammar for expressions sitting inside it, for example.

Going forward, we will occasionally come back to the Python grammar and associated documentation for authoritative answers about how code is interpreted.

As far as Python is concerned, the Python code we write first gets transformed into a tree representation. The parser that does this (or rather, an equivalent one) is also available as a Python module, in the `ast` (Abstract Syntax Tree) package. In LISP, we would directly have written that tree ourselves.

The most important statements we will have to know about are:

- Expressions (= statements that have a value).
- Import statements. These load a "module" / "software library" (which must be installed on the (virtual) machine):
  - `import numpy`
  - `import tensorflow as tf`
  - `from scipy import optimize as sopt`
- Assignments
  - At the "semantics" level, we need to discriminate between assignments that introduce a new variable and assignments that change the value of a variable.
  - Unlike in C, assignments do not have a value, but the grammar nevertheless allows us to chain them, as in `a = b = 0`.
  - The thing on the left hand side must designate some place that can hold a value. Destructuring is supported. The thing on the right must be an expression that has a single value, or an assignment (for chaining).
  - Examples
    - `x = 2 + 3 + f(y)`
    - `somelist[0] = foo.x = y = 7`
    - `(x, y, z) = (10, 20, 30)`
    - (...we will see quite a few more handy things in code examples later...)
- Function Definitions
  - There are both "named functions" and "unnamed functions".
  - A named function definition is a statement. Example:
    ```
    def squared(x):
      return x*x
    ```
    The form is:
    ```
    def {name}({...parameters...}):
      """ ... optional docstring ... """
      body_statement
      ... optionally, more body statements ...
    ```
  - Reaching a `return` statement exits the function, returning the value after `return`, if such a value given.
  - Reaching a bare `return` statement, or reaching end-of-body makes the function return the special value `None`.
  - "unnamed functions" / "lambda functions" are introduced as function-valued expressions of the form `lambda {parameters}: {expression}`.
    So, we can do `squared = lambda x: x*x`. The body is an expression, not a statement. There is no `return` here.
- Conditionals and loops.
  - `if/for/while/break/continue`
  - In time, we will see examples.
  - Surprisingly, these are less frequent/relevant in Python than many other languages, since "there often is a better way to do it".
- Exception handling
  - `try/except/finally` - only play a side role in this course.
- Class definitions
  - This basic course will mostly try to circumnavigate having to define classes. Pragmati-

cally, object-orientation is much about preventing authors from using global definitions for state-management - i.e. making state-management local by having parts of the relevant program-state (divided up in a meaningful way) be owned by different class instances ("objects") and manipulated through clearly defined pathways ("instance methods"). Often, a superior approach to state-management is *avoiding the need to manage state in the first place* - such as by focusing on designing programs in terms of data-transformations. In maths-heavy use cases (including Machine Learning), many relevant objects "have value semantics" - loosely, "behave in some sense like numbers", i.e. may have components akin to how a fraction has a numerator and a denominator, but should not be thought of as having any mutable properties - and this automatically does away with much of the need for state-management. The one place where we will obviously encounter stateful objects is in "trainable" components of a machine learning model. "Model training" is all about tweaking their internal state-parameters in a way that improves performance.

3.2 Python's "Data Model"

Python's Data Model in many ways resembles Common Lisp's, but its implementation resembles Perl more than any LISP system.

3.2.1 Key Aspects

- Variables are names that reference an in-memory value (or "object"). We must distinguish between the `name` (variable) and the value that is being referred to by a name. The variable is in itself not really an object, and there is only one situation where a variable shows up as a stateful ("container-like") entity.
- Every value in Python is an object in memory and has a memory address.

  These addresses are generally "hidden behind the scenes", we normally do not have to concern ourselves with these.

  The CPython interpreter actually exposes these addresses via the `id(.)` function. For every `x`, the value of `id(x)` is an integer such that `id(x) == id(y)` holds if and only if the variables `x` and `y` name the same in-memory-object. (If an object gets reclaimed after becoming provably-unreachable, a different object can have the same `id` later.)
- Every in-memory value/object has a clearly defined type. This type is a value/object itself, a "class object". The function `type(.)` maps an object to its type(-object).
- Terminology:
  - A `type` is the same as a `class`.
  - A class can be a "subclass" of other classes defined earlier.
  - "Is subclass of" is generally considered to be a reflexive(!) and transitive property, so, the term "subclasses of X" generally refers to the entire class hierarchy tree rooted at and including X.
  - If the type of an object `o` is `t`, then "`o` is an instance (a"class instance") of `t`", and also, if `t` is a subclass of `t0`, then `o` also is an instance of `t0`.
  - The `issubclass(t, t0)` function checks whether `t` is a (not necessarily proper) subclass of `t0`. Hence, `issubclass(type(x), type(x))` always evaluates to `True`, for every object `x`.

3.2.2 Important Types/Classes

Here, we are takling exclusively about the "dynamic types", the ones obtained by `type(.)`.

There also are "static type annotations", but we do not talk about these anywhere in this course.

- The type `int`.
  - Arbitrary-size integers (since Python3).
  - Literals (examples): `-5`, `100`, `1_000_000`, `0xff`.
- The type `bool`.
  - Subclass of `int`.
  - Literals: `True`, `False`.
  - Cannot be subclassed. This guarantees that `issubclass(type(x), bool)` implies `x is True or x is False` - there can only be these exact two in-memory objects of this type.
  - Used as an integer, `True` is equivalent to 1, and `False` is equivalent to 0, so (for example) `True + True = 2`.
- The type `float`.
  - IEEE-754 "binary64" floats.
  - Literals (examples): `1.23`, `-4.567e-12`.
  - "As everywhere else": 64-bit signed floating point numbers with just shy of 16 valid decimal digits. Valid exponents in decimal include the range (-300..300) and extend a little bit beyond that. There is minus-zero, positive-infinity, negative-infinity, and not-a-number. not-a-number is not equal to itself.
  - Note: `1.0/0.0` does not evaluate to float-infinity `float("+inf")`. Rather, evaluating this expression raises a `ZeroDivisionError`.
  - Related type: `complex`: pair of binary64 floats representing real and imaginary part of a complex number.
- The type `NoneType`.
  - There is only one in-memory object of this type.
  - Literal: `None`.
  - Cannot be subclassed.
  - Generally used in many Python APIs as a "poor man's substitute to express 'value can be missing'", but "category-theory-wise botched".
  - Terminology: in documentation, "an optional int" means: "this is an int instance, or the value `None`". (Likewise: "an optional string", etc.) Design problem: There is no sound notion of "an optional X" for arbitrary X, since "an optional optional int" does not work.
- The types `str` and `bytes`.
  - Informally, `str` instances represent text (such as: Unicode text), while `bytes` instances represent sequences of 8-bit bytes.
  - Unlike C/Java/etc, there is no separate `character` type `char`. In Python APIs, characters are represented by length-1 strings.
  - Literals
    - str: `"abc"`, `'def'`, `'''ghijk'''`, `"""xyz"""`
    - bytes: `b"abc"`, `b'def'`, etc.
    - There are special-form literals such as raw-string literals for which the rules about how to interpret embedded backslashes are different.
    - Adjacent literals "get merged into one literal at parse time" (like in C or C++).
    - Indexable, and so technically "sequences". Iterating over a string iterates over its 1-letter-long substrings.

    - Closely related to `bytes`: `bytearray` - which we will not discuss here.
- The type `tuple`.
  - Tuples are ordered sequences holding a finite number of (arbitrary-type) objects.
  - If `type(x) is tuple`, then `len(x)` is the number of elements of the tuple, and `x[0]` ... `x[len(x)-1]` refer to the elements of the tuple. *Index counting always starts at zero.*
  - Literals (examples): `()`, `(7,)`, `(1, True)`, `(1, (2, 5))`, `(1, -2.0, 7,)`, `(True, None, "foo", (2, 3.0), b"bar", bool)`.
  - There can be trailing commas.
    Syntax wart: a 1-tuple *must* be written with a trailing dot, since `(7)` would be a parenthesized expression that evaluates to the `int` instance `7`.
  - "Immutable" in the sense that they do not permit changing object-state by assigning to slots: `mytuple[0] = 7` will raise a `TypeError` exception.
  - "Addition" is overloaded on tuples to mean "concatenation" (similary for strings), so: `(1, 2) + (3, 4) => (1, 2, 3, 4)`.
- The type `list`.
  - Like a tuple, a `list` is an ordered sequence holding a finite number of objects (which in principle can be arbitrary-type, but quite often will be homogeneous).
  - Lists "have internal state", we can change the contents of a list-object:
    - Clearing the list: `mylist.clear()`.
    - Replacing the k-th element: `mylist[5] = 7`.
    - Appending an element at the end: `mylist.append(10)`.
      (This uses "slack space" allocated in geometric progression, like C++'s `std::vector<T>`, so is amortized-O(1).)
    - (There are more object-state-mutating operations.)
  - These are not 'lists' in the LISP sense: Producing a new list from a given one that has one extra element at the front is an expensive operation with Python lists, but accessing the N-th element requires *constant* effort, rather than "O(N) effort".
  - Literals (examples): `[]`, `[1, 2, 3]`, `[(1, True), (2, False)]`, `[None, [1, 2, 3], None, [4, 5], None]`.
  - Since we can assign to lists, we can make a list "contain itself": `xs = [None, 7]; xs[0] = xs`.
    General principle: Having objects for which traversing their components can get us back to the original object is technically possible in Python, but really really should always be avoided! (LISP does not have much of an issue here.)
- The type `range`.
  - Represents a range of integers that has a length, a range, a start-value, and a step size.
  - Conceptually, behaves a lot like a length-k tuple of integers with constant spacing, but is more efficient to represent in memory, iterate over, etc.
  - Example: `range(0, 10, 1)` is the range of integers starting at 0 (inclusive), ending at 10 (exclusive), with step size 1.
    The common short-hand here is `range(10)`. (Common convention throughout: start-indices are inclusive, end-indices exclusive.)
- The types `set` and `frozenset`.
  - Both set- and frozenset-instances represent unordered collections of different objects.
  - Instances of `set` have mutable state, so can be changed (adding/deleting elements). Instances of `frozenset` are immutable.
  - Constraint: Elements of `set` and also `frozenset` must not be mutable, unless they "define equality in terms of identity". This course: no sets/frozensets with mutable-state

elements!
  - Set literals (examples): `{1, 2}`, `{(5, 6), (7, 8)}`.
  - Syntax wart: This is not an empty set, but an empty dict(ionary): `{}`. As for other types, we can get an "empty/zero/neutral value" by calling the type-object with no parameters: `set()`.
  - Frozenset examples: No literals, but can create frozensets by calling `frozenset` with an arbitrary collection as a parameter: `frozenset([1, 2, 3])`, `frozenset({"abc", "ghi"})`, `frozenset(range(7))`
  - Caution: This does not work (`1` is not a collection): `frozenset(1)`.
    This might do something unexpected: `frozenset('cat')` (evaluates the same way as: `frozenset(['a', 'c', 't'])`).
- The type `dict`.
  - Table of key-value pairs, "with fast access", mostly corresponds to C++ `std::unordered_map<K, V>`, or Java `HashMap<K, V>`, but technically, neither keys nor values need be type-homogeneous.
  - Mutable, has state, we can add and delete entries.
  - For keys, the same rules apply as for set-elements. Unlike JavaScript and Perl, keys need not be strings, can be complex objects.
  - When used as a collection, mostly treated like the collection of its keys.
  - Literals (examples):
    `{"A": 1, "a": 1, "B": 2, "b": 2}`,
    `{3: 9, -3: 9}`,
    `{(1, 2, 3): (3, 2, 1), (5, 6): (6, 5)}`.
- The type `function` - and the abstract base class `Callable`.
  - Functions that map input-parameters to output-values.
  - There are different types of function-like objects:
    * built-in functions
    * user-defined functions
    * class instance methods
    * class methods
    * instances of classes implementing a `__call__` method

    These all are instances of the abstract base type `collections.abc.Callable`.
  - More about them later.
- The type `type`.
  - Instances of the type `type` are type-objects, such as `int`, `str`, `tuple`, `type`.
  - (See above): For many such type-objects, using them as callables with zero arguments evaluates to an 'empty' instance:
    `str() -> ""`, `int() -> 0`, `list() -> []`, etc.
  - Generally, we can introduce 'derivatives' of most such types, even `int`. (Some restrictions for types that only admit a fixed set of literals, like `bool`.)
- The `numpy` (numerical Python) module's `numpy.ndarray` type.
  - Not a core Python datatype, but something introduced and owned by a (prominent, but still: entirely optional) library(/"module").
    Likewise, Python has no built-in "variable in a symbolic expression" data type, but modules like `sympy` provide that. (LISP would use a `symbol` here.)
  - The reference that describes the contracts in detail is: https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html
  - Name means "n-dimensional array", but as usual, we can save ourselves a lot of confusion

if we consistently avoid language like "this 1-dimensional ndarray holds a 20-dimensional vector" or "this 2d array describes a 3d rotation". Much better: "this 1-index ndarray holds a 20-dimensional vector", respectively "this 2-index array describes a 3d rotation". (This convention unfortunately has not been adopted widely yet.)
  - TensorFlow's (and JAX's) tensors are closely modeled after `numpy.ndarray`.
  - General principle: Python (at least the CPython implementation) is a slow bytecode-interpreter. NumPy operations are executed by fast compiled and optimized numerical code. So, emulating something (like matrix multiplication) that can be done with NumPy operations through Python is generally not a good idea.
  - A useful basic and mostly-correct mental model:
    * instances 'under the hood' hold on to some block of memory with data (typically numerical data) in it, and provide extra bookkeeping which allows us to address this block of data in various ways, typically via indexing, using zero or more indices.
    * So, this naturally contains arrays with zero indices ("scalars", holding a single number), one index ("vectors"), two indices ("matrices"), etc.
    * Each such `ndarray` has a `size` (the number of elements), a "number of dimensions" `ndim` (misnomer - means: number of indices), and a `shape` (describing the range for each index), as well as a numerical data-type (`dtype`), and, of course, `data`.
    * numpy `ndarray`s are mutable: We can change data by assigning. (Not so for the "tensors" in ML frameworks - we will see why).
    * Example: we would naturally want to represent the 4x4x4x4 numerical entries of the Riemann curvature tensor (ignoring all symmetry considerations) at a given spacetime event as a float [4, 4, 4, 4]-ndarray.
    * Example: a 800x600 RGB image might be represented as a [600, 800, 3]-ndarray with dtype=uint8 (1-byte integers in the range 0..255). If `imgdata` is such a `ndarray`, we would have:
      · `len(imgdata) == 600`
      · `imgdata.size == 600 * 800 * 3`
      · `imgdata.dtype == numpy.uint8`
      · `imgdata.shape == (600, 800, 3)`
      · `imgdata.ndim == 3`
    * Wart: if `a` is a numpy-ndarray, then it depends on whether `a` is a 0-index array or not whether we can iterate over it. For this reason, `ndarray` instances technically are not Python sequences (like lists or tuples), but nevertheless in many ways behave sequence-like.
    * Important operations with arrays include (we will encounter many examples later):
      · Indexing readjustments: slicing, striding, reshaping, etc.
      · "broadcasting" (generalization of "adding a vector to every row in a matrix").
      · "generalized Einstein summation" type operations.
      · Padding, tiling.
      · Nonlinear element manipulations (such as: clipping outlier values).
- "Generator-Expressions" / "Iterators".
  - Just like `function` in Python is a subtype of `Callable`, these are all subtypes of a never-encountered-directly "abstract type" `collections.abc.Iterator`.
  - There are different specific such types, including e.g. dictionary-key-iterators.
  - These objects have internal state and behave like a tissue dispenser, only that they dispense Python objects: We can keep pulling objects from them, one after another, until they are exhausted. In particular, we can map and iterate over them.

– Very important for doing computations on-demand.

Overall, generator expressions tend to show up all over the place with modern Python code. An important aspect to keep in mind is that they are stateful, so consuming a generator/iterator involves side effects. It is important to ensure that in Python code, reading the code could not ever lead to mistaken mental models about what side effects occur at execution time. This normally means that all side effects are expected to be documented. the state-changes that occur when consuming an iterator are a borderline-exception in the sense that these are typically not documented when it is obvious that items will be pulled from an iterator. However, this also generally means that if a generator/iterator gets passed to a callable that does not precisely specify how much the iterator will be advanced, it is not OK to keep using the same iterator-object post-call, for example by passing it on to another callable. An important callable that allows re-using its iterator-argument post-call is - obviously - the `next()` built-in function, which retrieves the next element and advances the iterator by one. This example shows different generator-"dispensers" for squares.

```
>>> squares = (x*x for x in range(1, 11))
>>> next(squares)
1
>>> next(squares)
4
>>> next(squares)
9
>>> type(squares)
<class 'generator'>
>>> list(squares)
[16, 25, 36, 49, 64, 81, 100]
>>> squares2 = (x*x for x in range(1, 11))
>>> list(squares2)
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
>>> all_squares = map(lambda x: x*x, itertools.count(1))
>>> next(all_squares)
1
>>> next(all_squares)
4
# ... could go on "forever" ...
# This is an equivalent form that uses generator-expression syntax:
>>> all_squares2 = (x*x for x in itertools.count(1))
```

3.2.3 Other types

Python has many more built-in types, including somewhat esoteric ones such as as `module`, `NotImplementedType`, `ellipsis`, various file-handle types, etc, but we will not concern ourselves with most of these here.

One type that is somewhat relevant still is the type `object`. Every value is an instance of `object`, and every type is a subtype of `object`. (So, the `object`-type-value is an instance of the type `object`.)

3.2.4 Exceptions

One important subclass hierarchy to be aware of however is rooted at the `Exception` class/type. Instances are typically used as an argument to a `raise` statement, which early-terminates a function in such a way that, rather than the function returning a value, in-process nested function calls will be aborted until a call frame is reached that handles the (error-like) exception.

A typical example is this code:

```
def is_valid_probability(p):
  """Returns whether `p` is a valid probability."""
  return 0 <= p <= 1  # chained-comparison, equivalent to: 0 <= p and p <= 1.


def bayesian_rule(p1, p2):
  """Combines independent probablities via Bayes's rule."""
  if not (is_valid_probability(p1) and is_valid_probability(p2)):
    raise ValueError('Both p1 and p2 need to be valid probabilities. '
                     f'Got: p1={p1!r}, p2={p2!r}')
  p12 = p1 * p2
  return p12 / (p12 + (1 - p1) * (1 - p2))
```

3.3 Python's "Execution Model"

A key principle for every programming language is "scoping", both in the concrete technical sense (see: https://en.wikipedia.org/wiki/Scope_(computer_science), and in a generalized sense.

A basic problem is "Which context tells us what a particular named entity is referring to"? Unsurprisingly, this problem also exists in mathematics - and there are commonly accepted conventions that resolve all related ambiguities.

In mathematics, let us consider the following definition of the function hypotenuse$(\cdot, \cdot)$:

$$\text{hypotenuse}(x, y) = \sqrt{x^2 + y^2}$$

Now, if we defined a function $g(x) := \text{hypotenuse}(x, 1) + \text{hypotenuse}(1, x)$, we would all agree on the meaning of that definition, despite this definition of g using a variable name (namely x) that also occurs in the definition of hypotenuse. Properly formalizing the rules of such variable-substitutions and variable-binding is - however - not completely trivial.

Clearly, if we want to have "components supporting synthesis", then having some such rules that govern name-scoping is essential for building larger and more complicated units out of small units. (Just as in mathematics.)

There is basically one programming language where a conscious design decision was made to ignore such "synthesizability" considerations and go with a single flat namespace - "there is just one thing around that can be called x". This then naturally makes it increasingly hard to keep track of things the larger programs get - typically reaching un-manageability at about 1000 lines. That language is BASIC (specifically, the classic form of BASIC, Dartmouth BASIC pre-v6).

With code, there are basically three situations where such questions about "where is this named thing coming from" are relevant:

- Lexical scope - meaning of variables inside functions.

  Example: This is the translation-operator (written with spurious parentheses for more clarity):

  `translated = lambda a: (lambda f: (lambda x: f(x-a)))`

  This is a specific translation: `shifted5 = translated(5)`.

  Questions such as "what provides the translation-parameter `a` in this `shifted5` function" are about "lexical scope".

- Dynamic scope - meaning of context-parameters while some particular function is being evaluated. This is typically about resource management or some other implicit context we are referring to. "While we are still processing this function's body (and we might have stepped into evaluating some inner function), the term 'X' shall mean..."

  - In Python, dynamic scope (more commonly called "dynamic context") is typically handled by "context-manager objects" that "know how to 'enter' and 'exit' some context", and sometimes via `try/finally` blocks. The general idea is: "we need to close/drop/remove something as control flow leaves a block, no matter how" and "as long as the block is being processed, the to-be-closed/removed thing is available."
  - A common case is "writing data to a file":

    ```
    def write_record(filehandle, record):
      # ... writing data ...
      # (while this function-body here is being processed,
      # the file opened by the caller is open, and will
      # have to be closed even if an error arises here that
      # aborts further processing of the caller's function-body.)

    def write_all_records(filename, records):
      with open(filename, 'wt') as h_out:
        for record in records:
          write_record(h_out, record)
    ```

  - With TensorFlow1, there is (for example) a 'session-context' that provides the association with some hardware that can do a numerical calculation. With TensorFlow2, a "gradient-tape" is generally handled by such dynamic context.

- "Attribute resolution" - "where does a particular property of an object come from"?

  - Example: The `Counter` class in the `collections` module is a subclass of `dict`. `collections.Counter([1, 1, 1, 2, 2]).items()` uses the `items` attribute provided by the `dict` superclass, but calling the `.most_common()` method uses the attribute provided by the `Counter` class.
  - Object-oriented programming is a lot about using class instances ("objects") as a systematic approach to state-management.
  - As explained earlier, for maths-oriented code, where the fundamental notions are related to "properties of value-to-value transformations", we can get quite far not concerning

ourselves with OO at all - "since there is basically no state in need of management". (This will be mostly our perspective for this course.)

3.3.1 How Python code gets executed

An early "caveat" is in order here: a TensorFlow2 `@tf.function` decorated function looks like Python code but actually is something very different. We will come back to that.

A basic working model of Python code execution is:

- A `def xyz(...):` statement introduces a variable `xyz` whose value is a callable. The part in parentheses specifies parameters, and the indented lines below are the function's body, which may start with a docstring that is handled in a special way (made visible to tools).
- If a callable gets called, `x = f(p, q)`, the parameters are first evaluated in left-to-right order. Then, a new "execution frame" (/ "stack frame") gets created, a "workspace" where the variables specified as parameters in the callable definition come to life, get assigned the provided parameters, and stay alive until execution of the body completes, either via:
  - Processing body-statements reaches the end of the callable's body...
  - ...or an explicit `return` statement...
  - ...or a raised 'exception' aborts execution of the body.
- "The execution frame is the in-memory representation of the workspace that belongs to this particular call": It is *not* a property of the callable-object, but a property of the call(!).
- Body statements get executed one-after-another. Some statements may have one or multiple (indented) blocks of sub-statements. This includes: looping statements, conditionals ("if"), "with"-context-statements, etc.
- While a callable's body executes, variables refer to this "execution frame" - if they are found there. We have to discriminate two cases:
  - Assignment-to-a-name will assign to an existing variable with that name in the current execution frame, if it already existed, or introduce a new name on the current frame.
  - Evaluating-a-name (as an expression) will evaluate to the variable with that name in the current execution frame, if it exists. Otherwise, it will try to resolve the name against the variable-bindings that were in use at the time / in the context the current function was defined.
  - There are ways to fine-tune the rules about inclusion of specific variables from outer frames, via the `global` and `nonlocal` keywords.
- Unlike many functional languages (and even TeX), CPython does not perform "tail call optimization". (Basically, if recursive calls only occur in the form `return recursive_call(...parameters...)`, i.e. we would only ever pass through the output from the recursive call to our own caller, it is possible to instead transform the code to clear the current call-frame and hard-jump into the called function's code, making its return our own return. Implementation-wise, it is however more appropriate to think of this in terms of "representing"return" as a function-call and passing it into the continuation-function, which may or may not be recursing. Python does not do this.)

  Instead, there is actually a (changeable) limit on how deeply function calls can be nested - given by `sys.getrecursionlimit()` (typically 1000 or so).

In comparison to other languages, Python is a bit unusual in that variables generally are not declared, but get introduced by simply assigning to them. This mostly is OK, but there are some strange-looking cases to be aware of. This code block illustrates the weirdness:

[ ]:
```
x = 10

def funny1(a):
  return x + a  # x refers to the "outer x" above.


def funny2(b, c, do_assign=False):
  if do_assign:
    x = b
  return x + c


print('funny1(7) evaluates to:', funny1(7))
print('funny2(2, 30, do_assign=True) evaluates to:',
      funny2(2, 30, do_assign=True))
print('module-global x now is:', x)

# Executing this actually would raise an UnboundLocalError exception (try it!):
#
## funny2(2, 50, do_assign=False)
```

```
funny1(7) evaluates to: 17
funny2(2, 30, do_assign=True) evaluates to: 32
module-global x now is: 10
```

If, in the above code, we commented out the `if do_assign: x = b` part, then `x` would refer to the global `x`, as in `funny1`.

Here, however, even in a call with `do_assign=False`, which does not execute the `x=b` statement, the mere existence of such a statement somewhere on the function body makes `x` a name that belongs to the local frame and is not to be resolved from an outer frame (this makes x a "local name"). But then, in the `return x + c` line, we try to obtain its value without ever having set that variable, so it is an "unbound local".

Importantly, Python supports "lexical closures", so if a function is introduced (via a `def` or a `lambda`) in such a way that its body refers to a variable in use on the execution frame on which the `def` or `lambda` occurred, then the so-defined function retains a tie to the variable in outer scope. So, we can do this:

[ ]:
```
# These two functions make_adder_v1 and make_adder_v2 are effectively equivalent:
def make_adder_v1(k):
  return lambda x: x + k

def make_adder_v2(k):
  def add_k(x):
```

```
    return x + k
  return add_k


# Examples
add7v1 = make_adder_v1(7)
add7v2 = make_adder_v2(7)

# There is a tiny difference in how these function-objects print:
print(add7v1, add7v2)

# ...but not in their behaviour:
print(add7v1(10), add7v2(10))
```

```
<function make_adder_v1.<locals>.<lambda> at 0x7a4d22f39bd0> <function
make_adder_v2.<locals>.add_k at 0x7a4d22f39b40>
17 17
```

The important thing to remember is: If control flow reaches a `lambda` or crosses a `def`, this creates a function-object which retains ties to variables it uses that come from the outer lexical scope - either the immediate frame within which the function was defined, or some outer lexical frame of that frame.

We have to be careful here - such functions-referring-to-context-variables are basically the one place where "variables show up as having a life as stateful entities". We can do this...:

[ ]:
```
def make_adder_reporter(initial):
  cell = [initial]
  def report():
    return cell[0]
  def add(x):
    cell[0] = cell[0] + x
  return (add, report)

(the_adder, the_reporter) = make_adder_reporter(1000)
print(the_reporter())
the_adder(10)
print(the_reporter())
the_adder(5)
print(the_reporter())
```

```
1000
1010
1015
```

...and this is an excellent way to get bitten badly:

[ ]:
```
def make_multipliers(start, end):
  multipliers = []
  for k in range(start, end):
```

```
    multipliers.append((k, lambda x: k * x))
  return multipliers

the_multipliers = make_multipliers(5, 10)

for (k, multiplier) in the_multipliers:
  print('k:', k, ', multiplier(10):', multiplier(10))

# Let us compare this to...


def make_multipliers_v2(start, end):
  multipliers = []
  def get_multiplier_func(k):
    return lambda x: k * x
  for k in range(start, end):
    multipliers.append((k, get_multiplier_func(k)))
  return multipliers

the_multipliers2 = make_multipliers_v2(5, 10)

print('######')
for (k, multiplier) in the_multipliers2:
  print('k:', k, ', multiplier(10):', multiplier(10))


# The difference is of course that in the 2nd case, the multiplier-function
# holds on to a parameter-variable `k` introduced on the outer execution-frame
# of a function that received the value of the loop-iterator variable and
# bound it to its own (owned-by-the-call-frame!) cell named `k`.
#
# In the 1st case, every multiplier-function references the same variable,
# which is the loop-iteration variable, so it keeps changing its value during
# iteration - and once iteration is finished, retains the final value.
```

```
k: 5 , multiplier(10): 90
k: 6 , multiplier(10): 90
k: 7 , multiplier(10): 90
k: 8 , multiplier(10): 90
k: 9 , multiplier(10): 90
######
k: 5 , multiplier(10): 50
k: 6 , multiplier(10): 60
k: 7 , multiplier(10): 70
k: 8 , multiplier(10): 80
k: 9 , multiplier(10): 90
```

3.4 Useful things to know

The above descriptions explained the skeletal structure of Python.

Looking forward to the rest of this course, there are some things that deserve special attention, since they can save us a lot of work that otherwise would be very tedious.

3.4.1 N-index Arrays (in their various incarnations)

We already touched the `numpy.ndarray` "numerical n-index array" data type. The TensorFlow `tf.Tensor` and also the JAX `jax.numpy.ndarray` class are closely modeled after this, but "come with extra strings attached that make computing fast gradients simple".

Indeed, the important characteristics of a "TensorFlow Tensor object" are much less about tensor arithmetics than they are about "strings attached that associate this object with its role in a calculation, described by a tensor-arithmetic graph".

Many ML practitioners try to "visualize" 3+ index tensors as "higher-dimensional coefficient schemes" that extend the "0d=scalar - 1d=vector - 2d=matrix" chain. This is in general mostly a useless distraction. An overall more useful perspective is to regard any such "n-index tensor/array" as a "table", which associates a particular tuple of indices with a value. So, overall, these "tensors" are conceptually much closer to tables in a relational database (such as a SQL database) than to "cubes and hypercubes of numbers". That is actually also what we do in physics.

The following code cell takes us through a quick tour of some common `numpy.ndarray` operations - using some familiar tensors. The `tf.Tensor` and `jax.numpy.ndarray` classes typically provide very similar functionality, often using the same names.

[ ]:
```
import numpy   # Loading the `numpy` module.
import pprint  # Also loading the "pretty-printing" module.

pauli_matrices = numpy.array(
    [[[1, 0],
      [0, 1]],  # 1
     ###
     [[0, 1],
      [1, 0]],  # sigma_x
     ###
     [[0, -1j],
      [1j, 0]],  # sigma_y
     ###
     [[1, 0],
      [0, -1]],  # sigma_z
    ], dtype=complex)

# Rather than defining the gamma matrices in a simple and uniform fashion,
# let us showcase a few different approaches to set these up.
# With numpy.ndarray, the focus sometimes is on mutation, but tf.Tensor
# and jax.numpy.ndarray are immutable, so here we also de-emphasize mutation
# operations, favoring "mathematical definitions"
```

```
gamma0 = numpy.diag([1, 1, -1, -1])
gamma1 = numpy.diag([-1, -1, 1, 1])[::-1, :]  # "reverse-ordered rows".
gamma2 = 1j * numpy.diag([-1, 1, 1, -1])[::-1, :]
gamma3 = numpy.einsum('AB,ab->AaBb',
                      -1j * pauli_matrices[2],
                      pauli_matrices[3]).reshape(4, 4)

gammas = numpy.stack([gamma0, gamma1, gamma2, gamma3], axis=0)

gamma5 = gamma0 @ gamma1 @ gamma2 @ gamma3

print('=== gammas ===')
pprint.pprint(gammas.tolist())
print('gamma5:', gamma5.tolist())

gamma_mu_nu = numpy.einsum('amn,bnp->abmp', gammas, gammas)
clifford_rhs = gamma_mu_nu + numpy.einsum('abmn->bamn', gamma_mu_nu)
eta_mu_nu = numpy.diag([1, -1, -1, -1])
assert numpy.allclose(clifford_rhs,
                      # This uses "broadcasting" - indexing with a
                      # `numpy.newaxis` index adds an extra range-1 index
                      # to a n-index array in the corresponding place,
                      # and operations such as element-wise(!) multiplication
                      # of a tensor with another "repeat along the range-1
                      # indices". The product in the two lines below
                      # is the same as
                      # `einsum('ab,cd->abcd', 2*eta_mu_nu, eye(4))`, where
                      # `eye(.)` returns an identity matrix.
                      2 * eta_mu_nu[:, :, numpy.newaxis, numpy.newaxis] *
                      numpy.eye(4)[numpy.newaxis, numpy.newaxis, :, :]), (
        'Clifford algebra has a bug.')

P_L = 0.5 * (numpy.eye(4) - gamma5)
P_R = 0.5 * (numpy.eye(4) + gamma5)

print('=== tr(gamma_mu gamma_nu) ===')
pprint.pprint(numpy.einsum('abmm->ab', gamma_mu_nu).tolist())
```

```
=== gammas ===
[[[(1+0j), 0j, 0j, 0j],
  [0j, (1+0j), 0j, 0j],
  [0j, 0j, (-1+0j), 0j],
  [0j, 0j, 0j, (-1+0j)]],
 [[0j, 0j, 0j, (1+0j)],
  [0j, 0j, (1+0j), 0j],
  [0j, (-1+0j), 0j, 0j],
  [(-1+0j), 0j, 0j, 0j]],
```

```
 [[0j, 0j, 0j, (-0-1j)],
  [0j, 0j, 1j, 0j],
  [0j, 1j, 0j, 0j],
  [(-0-1j), 0j, 0j, 0j]],
 [[0j, 0j, (-1+0j), 0j],
  [0j, 0j, 0j, (1+0j)],
  [(1+0j), 0j, 0j, 0j],
  [0j, (-1+0j), 0j, 0j]]]
gamma5: [[0j, 0j, 1j, 0j], [0j, 0j, 0j, 1j], [1j, 0j, 0j, 0j], [0j, 1j, 0j, 0j]]
=== tr(gamma_mu gamma_nu) ===
[[(4+0j), 0j, 0j, 0j],
 [0j, (-4+0j), 0j, 0j],
 [0j, 0j, (-4+0j), 0j],
 [0j, 0j, 0j, (-4+0j)]]
```

3.4.2 "Python as a Query Language"

The "machine oriented" approach to programming tends to focus on questions such as "where is this value stored?" and "what parts of the program have permission to mutate it?".

Mutations play a big role, but unfortunately do not nicely align with how we normally structure analysis of complicated relations in mathematics - since in maths, there generally is no notion of "x was defined as follows, and under these specific circumstances, we henceforth change its meaning to mean z".

So, when we have to reason about (perhaps even prove theorems about) code containing mutations ("imperative programs"), we need some heavy handed and unwieldy machinery. (A widely used grad course textbook on the subject is Gougen & Malcolm, Algebraic Semantics of Imperative Programs [5].)

Overall, we are usually in a better position if we align code structure with a constructive mathematical construction of our objective, written up the way we usually write up maths. While this here is purely a matter of improving understandability, the same point also has emerged in research on compilers and microprocessors. Even down at the microprocessor level, if - for example - machine code says to store an intermediate quantity in register R7, but this register is used in machine code to hold another - completely independent - intermediate quantity later, calculations that could be done concurrently nominally cannot because they want to use the same storage location for an intermediate result. So, modern microprocessors have "register renaming" logic to eliminate such "false data dependencies". At the compiler level, human-written code first gets parsed (we have seen how that works), and then translated into some intermediate representation whose 'variables' are define-once-read-only and never get reassigned to - "SSA form". Again, the point is that such code is easier to reason about, in particular for exploring opportunities to apply performance-increasing transformations.

Overall, list/set/dict comprehensions and also generator expressions are very powerful Python language features that allow us "to focus on the What and not the How".

Let us look at the following piece of Python code. This "inverts a mapping specified by a table" (which here must be injective).

[ ]:
```
def inverted_mapping(table):
  result = {}
  for key, value in table.items():
    result[value] = key
  return result

name_by_digit = {1: 'one', 2: 'two', 3: 'three', 4: 'four', 5: 'five',
                 6: 'six', 7: 'seven', 8: 'eight', 9: 'nine', 0: 'zero'}

digit_by_name = inverted_mapping(name_by_digit)
print(digit_by_name)
```

```
{'one': 1, 'two': 2, 'three': 3, 'four': 4, 'five': 5, 'six': 6, 'seven': 7,
'eight': 8, 'nine': 9, 'zero': 0}
```

The `inverted_mapping` function's body is imperative: "First, create a new empty-dictionary object, which will be turned into the result by mutation operations. Then, loop over the key-value pairs of the input dictionary, and for every pair, add a new entry to the result-dictionary, with the role of key and value swapped. Finally, return the result" - one command after another.

The alternative forms below use special Python syntax - a "dictionary comprehension" - to express this idea in a way that can be read without keeping track of intermediate state.

[ ]:
```
inverted_mapping_v2 = lambda table: {value: key for key, value in table.items()}


# Equivalently, with `def`:
def inverted_mapping_v3(table):
  return {value: key for key, value in table.items()}
```

There are some extra subtleties here, such as "what guarantees does Python give about the order in which `table.items()` produces the key-value pairs from the dictionary - and what then are the corresponding guarantees for the result dictionary?". However, in those many situations where we need not concern ourselves with such aspects, we can simply forget about them and focus on what we need to reason about - just as it should be!

On the general topic of "reasoning about the behavior of Python code", there unfortunately are some design warts that make this more complicated than it ought to be, such as `sys.getrecursionlimit()` generally spoiling theorems about code with necessary qualifications such as "form A is equivalent to form B, unless evaluating form B breaks the recursion-depth limit". The author still has hope that Python can evolve in the direction of making it easier to reason about code by eliminating such doubtful design decisions, but this will only happen if the academic community (and also the computer security community) makes a strong point about how important this is!

This approach - focusing on the logic of computations rather than the steps - is known as "declarative programming". Python offers quite a few tools that can be used to great advantage especially for those parts of a program that are not about "number-crunching", including of course "getting the data", and "wiring up the components of a larger application".

The code cell below illustrates this with a few examples.

[ ]:
```
print('\n### Sum-of-divisors of a number (brute-force) ###')

def sum_of_divisors(n):
  return sum(d for d in range(1, n) if n % d == 0)

for n in (3, 4, 5, 6, 7, 8, 9, 10, 28, 2*248, 8128):
  print(n, sum_of_divisors(n))

print('\n### Sorting version-number strings by release-order ###')

linux_kernel_versions = ['2.0', '4.3', '2.2', '4.20', '2.4',
                         '2.6', '5.4', '2.6.11', '2.6.39']

def release_order_sorted_version_strings(versions):
  # When sorting an iterable, we can optionally specify a key-function.
  # If this is provided, elements are sorted according to monotonically
  # increasing value of the key-function. Magnitude-comparison on tuples
  # and also on lists uses lexicographic ordering.
  return sorted(versions, key=lambda v: [int(x) for x in v.split('.')])

print(release_order_sorted_version_strings(linux_kernel_versions))

print('\n### Finding integers >0 that are not a sum of k>1 consecutive '
      'integers >0 ###')

import itertools

def numbers_that_are_not_a_sum_of_consecutive_numbers(limit):
  range_sum = lambda n: n * (n + 1) // 2  # `//` is int/int->int division.
  sums_of_consecutive_numbers = {  # set-comprehension!
      range_sum(upper) - range_sum(lower - 1)
      # Note: itertools.combinations(('a', 'b', 'c'), 2) ->
      # generator yielding elements: ('a', 'b'), ('a', 'c'), ('b', 'c').
      for lower, upper in itertools.combinations(range(1, limit + 1), 2)
  }
  return set(range(1, limit+1)) - sums_of_consecutive_numbers

print(numbers_that_are_not_a_sum_of_consecutive_numbers(1000))

print('\n### Brute-force counting involutions ###')
import pprint

def number_of_involutions(permutation_length):
  return sum(all(p[p[k]] == k for k in range(permutation_length))
             for p in itertools.permutations(range(permutation_length)))

pprint.pprint({n: number_of_involutions(n) for n in range(10)})
```

```
# Let's try to do something OS related using this approach.
print('\n### All running processes on the current VM that have the same '
      'parent process as the current process ###')
import glob, re, os

def all_processes_with_this_parent_pid(wanted_ppid):
  def get_name_and_pid_and_ppid(pid):
    with open(f'/proc/{pid}/status') as h:
      match = re.match(r'(?sm)Name:\s*(\S*).*?^PPid:\s*(\S+)', h.read())
      return (match[1], pid, int(match[2]))
  return sorted((name, pid)
                for name, pid, ppid in map(get_name_and_pid_and_ppid,
                                           (d for d in os.listdir('/proc')
                                            if d.isdigit()))
                if ppid == wanted_ppid)

print(all_processes_with_this_parent_pid(os.getppid()))
# (It so turns out that the current process has no siblings. Let's check.)
print('---')
!pstree --ascii
```

```
### Sum-of-divisors of a number (brute-force) ###
3 1
4 3
5 1
6 6
7 1
8 7
9 4
10 8
28 28
496 496
8128 8128

### Sorting version-number strings by release-order ###
['2.0', '2.2', '2.4', '2.6', '2.6.11', '2.6.39', '4.3', '4.20', '5.4']

### Finding integers >0 that are not a sum of k>1 consecutive integers >0 ###
{32, 1, 2, 64, 4, 128, 256, 512, 8, 16}

### Brute-force counting involutions ###
{0: 1, 1: 1, 2: 2, 3: 4, 4: 10, 5: 26, 6: 76, 7: 232, 8: 764, 9: 2620}

### All running processes on the current VM that have the same parent process as
the current process ###
[('python3', '1795')]
```

```
---
docker-init-+-node-+-colab-fileshim.
            |      |-dap_multiplexer---4*[{dap_multiplexer}]
            |      |-jupyter-noteboo-+-python3-+-pstree
            |      |                 |         `-13*[{python3}]
            |      |                 `-6*[{jupyter-noteboo}]
            |      |-language_servic-+-node---7*[{node}]
            |      |                 `-7*[{language_servic}]
            |      |-oom_monitor.sh---sleep
            |      |-python3
            |      `-10*[{node}]
            |-python3---6*[{python3}]
            `-run.sh---kernel_manager_---4*[{kernel_manager_}]
```

3.4.3 General principles for good code readability and maintainability

The principles spelled out below may sound self-evident, but there is a lot of code around that violates them. By making "sticking to these principles a habit", one can do a lot for one's code quality and maintainability.

- The documentation of an API (such as: a function to be used by others) is an integral part of the API, and considered to be "a binding contract" - "if you use this thing as follows, then that is the guaranteed behavior."
  - Uses of the API must not violate the contract.
  - It is not OK to rely on accidental implementation details that are not part of the documented API contract.
  - A subclass contract can refine but must not override parts of a parent class contract.
  - Contracts must not contradict themselves.
- Error-checks must match error messages precisely! No "Checking one thing and reporting something different."
- Never change global state you do not own. (Such as: re-seeding a global number generator, changing floating point rounding mode, etc.) Do not use global state in your designs, unless absolutely unavoidable.
- Constrain side effects and state-mutation to the narrowest possible context - whatever can be described and coded out as a simple value-transformation "that feels mathsy" should be written that way.
- Code comments should not repeat what the code already says, but are very appropriate to explain relevant subtleties.
- Simple questions about the code should have simple answers. (Especially: "What's this variable?")
- Spend some effort on naming things properly.
- Think about possible sources of concept-confusion for the reader - and eliminate them with well-designed terminology.

  (This includes for example: Don't ever call a 20x20 matrix "two-dimensional"!)

- No "hope and pray" coding that uses checks/properties which generally hold, despite having counterexamples.
- No "inferring properties from examples only" - both when writing code and when documenting APIs.
- Think about the relevant ("natural") invariants that the concept entities your design is introducing should satisfy.
- Make everything reproducible wherever you can! If you ever are in a situation of "this result that here showed up by chance is very remarkable", and your computer crashes the next minute, you want to be able to reproduce that "chance discovery"!
- Design should be explainable -"everything should be there for a good reason" (and we should at least be able to articulate that reason why the design decision makes sense).

Importantly, "designing with mathematical transformations as key entities" and "keeping state management minimal" also helps a lot with making code testable!

One of the currently most under-valued and under-appreciated Software Engineering skills (relative to its importance) is "defining the language to talk about the (typically, complicated) problem". This "language" may consist of classes, functions, other code parts, but also - very importantly - new terms you invented to describe the design in its documentation. A typical solid Computer Science PhD thesis is about "defining a large language that includes natural-language terminology and typically also code definitions to make a relevant real world problem manageable".

When working on a complicated project, it is perfectly natural to start out with a rough mental model where some ideas are off - there may be notions of some important concepts and aspects, but their role may be under-appreciated at first, or "otherwise a bit off". (Example: in a concurrent database, should operations have event-timestamps, or be associated with time-intervals over which the operation will have affected database state? Or: Are human-readable error messages appropriate here? If something else processes these, and then expects some specific originally-intended-for-humans text, there are issues with localization (translating to other languages), and also issues with even just fixing typos. Should the design be adjusted to provide both an error-message-for-humans and also error-code-for-programs?)

The important competency is to "make progress by exploring and playing with the subject matter clay, shaping things in different ways - and throwing away many design ideas, progressing towards more appropriate notions that manage to capture the key aspects in a useful way".

It is natural and expected that, during this "playing with the clay" process of language-finding, we explore a lot and throw away a lot. In that situation, it matters a lot to clearly document the role of every piece of code - there is a world of a difference between code that describes itself as "this is merely for exploring a weird idea, and is kept here to document how far we got and what problems we ran into" and code that describes itself as "this is a software component that was diligently engineered to for-use-in-production quality".

"**Excellent engineering is mostly language design**" - introducing a suitable way to talk about a problem, analyze it, share insights with other experts, and explain how solutions are built, so others can understand, use, and adjust them.

Participant Exercise: What are relevant "terminology inventions" in nuclear engineering that were invented to capture relevant aspects of how radiation interacts with people, what were the precursors of some of these modern terms, and what were the drawbacks that were identified as a reason to replace them? How are product design decisions tied to that terminology?

4 Important References

- Python Reference - https://docs.python.org/3/reference/
- Python Grammar - https://docs.python.org/3/reference/grammar.html
- Python Operator Precedence table: https://docs.python.org/3/reference/expressions.html#operator-precedence
- Python Standard Library - https://docs.python.org/3/library/

---

5 Proposed Candidate Python Coding Exercises

1. Find a square number which is the sum of consecutive square numbers $1+4+9+\ldots$
2. Find an integer solution $(a, b) \in \mathbb{N}^2$ to $13a^2 + 1 = b^2$.
3. Implement a function that computes the number of ways to choose k elements out of n with replacement.
4. Implement a function, which for two pairs of spherical coordinates (ϑ, ϕ) computes their angular distance. This should never fail, and produce numerically accurate results for small-angle distances.
5. Implement a function which, when given a collection of `k` (complex-float) matrices of shape $N \times N$, determines if all commutators also lie in the span of these matrices, and if so, computes the structure constants ${f_{ab}}^c$ satisfying $[M_a, M_b] = {f_{ab}}^c M_c$.
6. Implement a function which returns a (fairly sampled) random element of $SO(d, \mathbb{Z})$.
7. Implement a function which, when given a $\mathbb{R}^N \to \mathbb{R}^N$ function f plus its Jacobian, as a $\mathbb{R}^N \to \mathbb{R}^{N \times N}$ function, finds a zero f via multidimensional Newton iteration.
8. Implement a function which, when given a (small) graph's adjacency matrix, determines the number of connected components.
9. Implement a function that maps a list of numbers to a function evaluating the polynomial with the given coefficients (using Horner's scheme).
10. Implement a function that determines whether a tuple of integer indices represents a permutation (i.e. contains every integer from 0 (included) up to tuple-length (excluded) exactly once).

5.1 Addendum: Graph structure of Python in-memory objects

(These deeper details may be useful knowledge in some situations, especially when dealing with foreign function interfaces, but can be skipped on a ML-focused course if time is short.)

Every Python value has an underlying in-memory representation as a `struct PyObject` instance. In general, in bytecode-interpreter implementations of languages such as Perl or Python, it is not uncommon to see the corresponding object-pointers to point to a struct-instance that is the not-first element of a larger struct, so that there can be extra in-memory bookkeeping information that sits right before the object-instance. In general, functional languages typically use some type-tagging scheme for representing values in-memory that avoids having to go through a memory-reference for small integers, characters, and other such simple objects, but this is not the case for the CPython interpreter in version 3.11 or earlier.

Both for Perl and Python, in-memory objects use a reference-counting approach to memory-management, which means that an object knows the total number of times it is being referred to by in-memory objects, and can be reclaimed when that count drops to zero - which means that the object became provably-unreachable (given that the other code is correct w.r.t. memory handling). Such a simple reference-counting approach avoids the problem of requiring some sophisticated Garbage Collection (as just about every functional programming language and also e.g. `Java`, `JavaScript`, `C#` etc. have it), but runs into problems when data structures contain circular references - these can make objects unreclaimable. Since in a non-lazy language like Python, circular references can only be produced by resorting to object-mutation and we mostly can get away with data-transformation focused code structure in ML, we will subsequently not have to concern ourselves with in-memory representation details. Still, it should be pointed out that if we were to liberally use object-mutation, this would come with subtle potential pitfalls. Each of the definitions below introduces a `4x4` matrix-like data structure, a list of four lists of four identical floating point numbers. However, when mutating the top-left element of each of these matrices by assigning to it, it shows that in the second and third case, the list of row-lists references identically the same row-object four times, so making a mutation of that row-object affects what we see on every row-slot of the matrix.

[2]:
```
m1 = [[5.0, 5.0, 5.0, 5.0],
      [5.0, 5.0, 5.0, 5.0],
      [5.0, 5.0, 5.0, 5.0],
      [5.0, 5.0, 5.0, 5.0]]


m2 = [[7.0, 7.0, 7.0, 7.0]] * 4

m3 = [[8.0] * 4] * 4

m4 = [[9.0] * 4 for _ in range(4)]

m1[0][0] = 2.0
m2[0][0] = 2.0
m3[0][0] = 2.0
m4[0][0] = 2.0

print('--- m1 ---\n', m1, sep='')
print('--- m2 ---\n', m2, sep='')
print('--- m3 ---\n', m3, sep='')
print('--- m4 ---\n', m4, sep='')
```

```
--- m1 ---
[[2.0, 5.0, 5.0, 5.0], [5.0, 5.0, 5.0, 5.0], [5.0, 5.0, 5.0, 5.0], [5.0, 5.0,
5.0, 5.0]]
--- m2 ---
[[2.0, 7.0, 7.0, 7.0], [2.0, 7.0, 7.0, 7.0], [2.0, 7.0, 7.0, 7.0], [2.0, 7.0,
7.0, 7.0]]
--- m3 ---
[[2.0, 8.0, 8.0, 8.0], [2.0, 8.0, 8.0, 8.0], [2.0, 8.0, 8.0, 8.0], [2.0, 8.0,
8.0, 8.0]]
--- m4 ---
[[2.0, 9.0, 9.0, 9.0], [9.0, 9.0, 9.0, 9.0], [9.0, 9.0, 9.0, 9.0], [9.0, 9.0,
9.0, 9.0]]
```

The following code cell (which is independent of the rest of this notebook and can be copy-pasted and adjusted/extended independently) contains a bare-bones skeleton of a graph-render of the memory object reference structure of a composite Python value.

[3]:
```
!apt -qqq install graphviz

import io
import os
import re
import subprocess
import sys
import time
import IPython.display


_DOT_HEADER = """
digraph pyobject {
  fontname="Palatino,Roboto,Helvetica"
  node [fontname="Palatino,Roboto,Helvetica"]
  edge [fontname="Palatino,Roboto,Helvetica"]
  graph [
    rankdir = "LR"
  ];
  node [
    fontsize = "12"
    shape = "record"
  ];
  edge [
  ];
  obj [label="OBJECT"];
  obj -> n0:obj;
"""


def _slots_str(num_slots):
```

```
  return '|'.join(f'<s{n}>' for n in range(num_slots))


def _type_name(obj):
  return re.sub(r'[^\w ]', '?', type(obj).__name__)


def _render_tuple(obj):
  return (f'[shape=record,label="<obj>T:
 ↪{_type_name(obj)}|{_slots_str(len(obj))}",'
          'style=filled,fillcolor="#94eafe"]')


def _render_list(obj):
  return (f'[shape=record,label="<obj>T:
 ↪{_type_name(obj)}|{_slots_str(len(obj))}",'
          'style=filled,fillcolor="#f2e7c0"]')


def _render_set(obj):
  return (f'[shape=record,label="<obj>T:
 ↪{_type_name(obj)}|{_slots_str(len(obj))}",'
          'style=filled,fillcolor="#bbbbee"]')


def _render_frozenset(obj):
  return (f'[shape=record,label="<obj>T:
 ↪{_type_name(obj)}|{_slots_str(len(obj))}",'
          'style=filled,fillcolor="#8899ee"]')


def _render_dict(obj):
  slots = '|'.join('{<sk%s>|<sv%s>}' % (n, n) for n in range(len(obj)))
  return (f'[shape=record,label="<obj>T:{_type_name(obj)}|{slots}",'
          'style=filled,fillcolor="#a5ff9c"]')


def _render_int(obj):
  return (f'[shape=record,label="<obj>T:{_type_name(obj)}|{obj}",'
          'style=filled,fillcolor="#eebbbb"]')

def _render_float(obj):
  return (f'[shape=record,label="<obj>T:{_type_name(obj)}|{obj}",'
          'style=filled,fillcolor="#aaaaff"]')


def _render_str(obj):
```

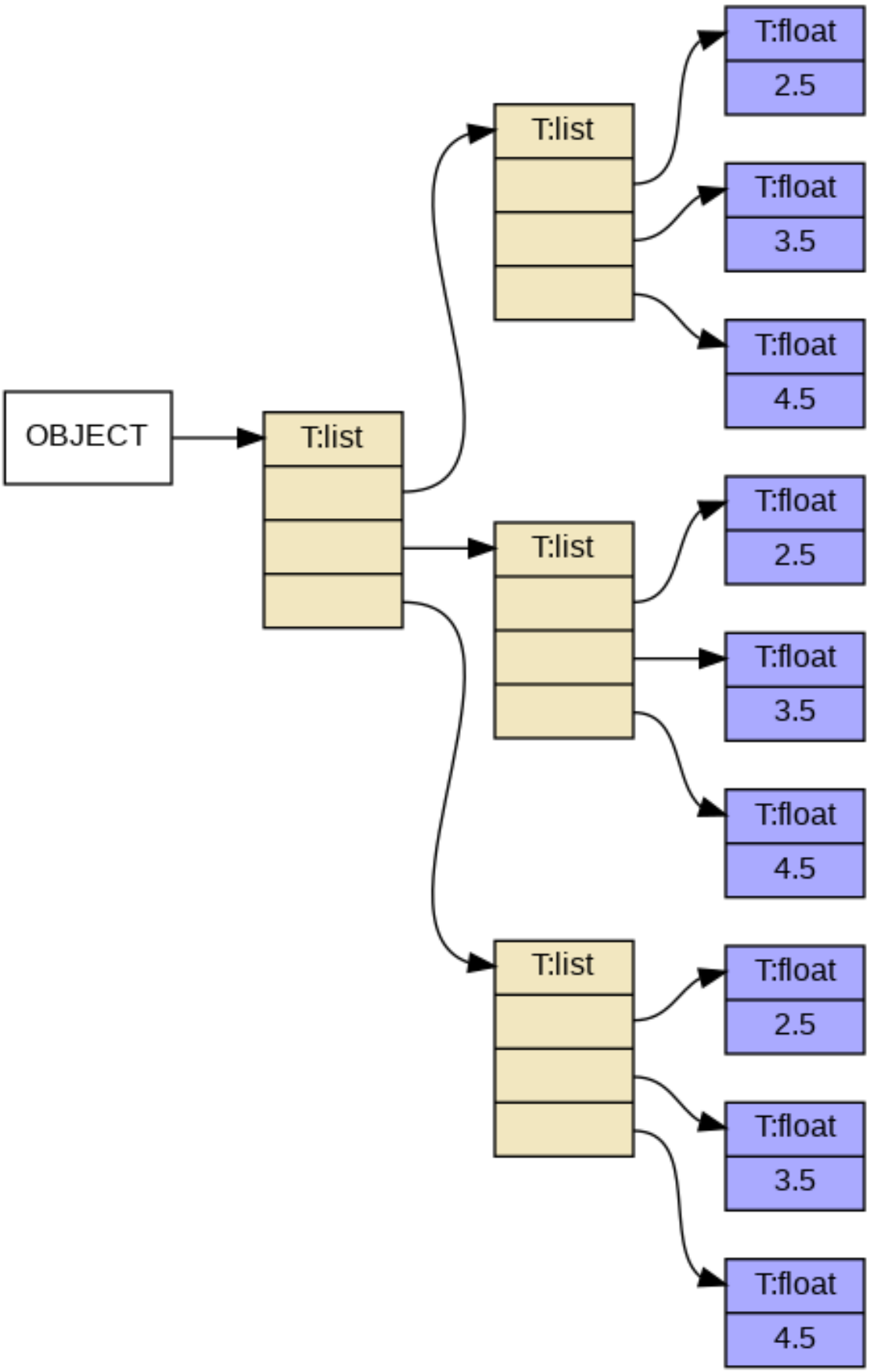
```
  # String content might clash with graphviz syntax, so we have to be careful␣
 ↪here.
  translated = re.sub(r'[^-A-Za-z0-9_ .,:;%!?@#$\[\]\{\}]', '[?]', obj)
  return (f'[shape=record,label="<obj>T:{_type_name(obj)}|{translated}",'
          'style=filled,fillcolor="#faffaf"]')


def _render_none(obj):
  return (f'[shape=record,label="<obj>T:{_type_name(obj)}|{obj}",'
          'style=filled,fillcolor="#aaaaaa"]')


def _render_unknown(obj):
  return (f'[shape=record,label="<obj>T:{_type_name(obj)}|(Unknown contents)",'
          'style=filled,fillcolor="#88ee88"]')


_RENDERER_BY_TYPE = {
    int: _render_int,
    float: _render_float,
    str: _render_str,
    tuple: _render_tuple,
    list: _render_list,
    set: _render_set,
    dict: _render_dict,
    frozenset: _render_frozenset,
    type(None): _render_none,
}


def as_graphviz(obj):
  """Returns a .dot string for a graph visualizing the object."""
  # 'Canonicalized id' - does not leak memory layout information.
  cid_by_id = {}
  #
  def get_cid(obj):
    return cid_by_id.setdefault(id(obj), len(cid_by_id))
  #
  node_by_cid = {}
  rendered_by_cid = {}
  edges = []
  #
  def walk(node):
    cid = get_cid(node)
    if cid in node_by_cid:
      # Properly handling reference cycles.
      return
```

```
    node_by_cid[cid] = node
    node_type = type(node)
    rendered_by_cid[cid] = _RENDERER_BY_TYPE.get(node_type,
                                                 _render_unknown)(node)
    if issubclass(node_type, (list, tuple, set, frozenset)):
      for n, child in enumerate(node):
        walk(child)
        edges.append(
          f'n{cid}:s{n} -> n{get_cid(child)}:obj')
    elif issubclass(node_type, dict):
      for n, (key, val) in enumerate(node.items()):
        walk(key)
        walk(val)
        edges.append(
          f'n{get_cid(key)}:obj -> n{cid}:sk{n} [dir=back]')
        edges.append(
          f'n{cid}:sv{n} -> n{get_cid(val)}:obj')
  #
  walk(obj)
  result = io.StringIO()
  result.write(_DOT_HEADER)
  for cid, rendered_node in sorted(rendered_by_cid.items()):
    result.write(f'  n{cid} {rendered_node};\n')
  for edge in edges:
    result.write(f'  {edge};\n')
  result.write('}\n')
  return result.getvalue()


def pygraph(obj, directory='.'):
  tag = int(time.time() * 1e9)
  dot_text = as_graphviz(obj)
  file_stem = os.path.join(directory, f'pygraph_{tag}')
  file_dot = file_stem + '.dot'
  file_png = file_stem + '.png'
  with open(file_dot, 'wt') as h:
    h.write(dot_text)
  done = subprocess.run(['dot', '-Tpng', file_dot], capture_output=True)
  with open(file_png, 'wb') as h:
    h.write(done.stdout)
  return IPython.display.Image(file_png)
```

With this we can visualize values such as list-of-lists matrices in a way that shows the structural difference:

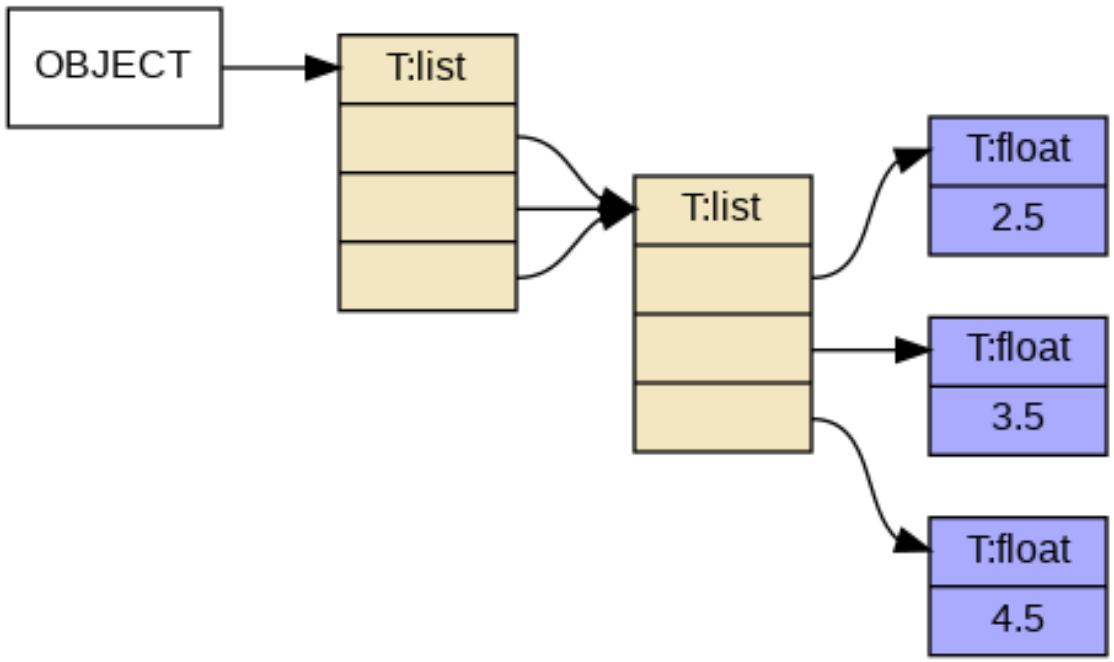
[7]:
```
pygraph([[x + 1.5 for x in range(1, 4)] for _ in range(1, 4)])
```

[7]:


OBJECT
T:list
T:list
T:float
2.5
T:float
3.5
T:float
4.5
T:list
T:float
2.5
T:float
3.5
T:float
4.5
T:list
T:float
2.5
T:float
3.5
T:float
4.5

[8]:
```
pygraph([[x + 1.5 for x in range(1, 4)]] * 3)
```

[8]:



6 Derivatives, Numerical Optimization, and Machine Learning

6.1 Why we are looking into this

- ML "Model training" generally is a form of numerical optimization - "minimizing the risk of a seriously off answer".
- Common ML situation: "very many" optimization parameters (ballpark ~10^4 to 10^9).
- Being able to do numerical optimization with >10 parameters is very useful in itself.
- Many approaches to numerical optimization of nonlinear functions use gradients.
- So, we want to retrace/understand the steps that led to insights on how to efficiently compute gradients even for $\mathbb{R}^{1\,000\,000} \to \mathbb{R}$ objective functions.

But let us start simple with something we know - also to get some Python practice: Derivatives.

[ ]:
```
# Importing the relevant Python libraries ("modules").
import math
import pprint
import time

import matplotlib
from matplotlib import pyplot
```

```
import numpy
import scipy.optimize
import tensorflow as tf

# Larger figures by default.
matplotlib.rcParams['figure.figsize'] = (16, 8)
```

First task: Given a simple function ("black box" - we can evaluate it at individual points, but not inspect its definition) "where all relevant scales (function value, gradient, curvature, etc.) are of magnitude 1", let us look into numerically computing the derivative at a given point.

[ ]:
```
def derivative_v0(f, x0, eps=1e-6):
  """Estimates f'(x0) via finite differences.

  Args:
    f: Callable, the R->R function we want to compute the derivative of.
    x0: The position where to compute the derivative.
    eps: Step size.

  Returns:
    `(f(x0+eps) - f(x0)) / eps`, as a numerical approximation
    to the derivative.
  """
  return (f(x0 + eps) - f(x0)) / eps


def derivative_v1(f, x0, eps=1e-6):
  """Estimates f'(x0) via finite differences (symmetric difference).

  Args:
    f: Callable, the R->R function we want to compute the derivative of.
    x0: The position where to compute the derivative:
    eps: Step size.

  Returns:
    `(f(x0+eps) - f(x0-eps)) / (2*eps)`, as a numerical approximation
    to the derivative.
  """
  return (f(x0 + eps) - f(x0 - eps)) / (2 * eps)


def demo1(position=2.0, derivative=derivative_v0):
  f = lambda x: x**3
  for epsilon in (0.01, 1e-3, 1e-4, 1e-5, 1e-6, 1e-7, 1e-8, 1e-9, 1e-10):
    df_dx = derivative(f, position, eps=epsilon)
    print(f"eps={str(epsilon):6s}: f'({position}) = approx. {df_dx}")
```

7 Exploring Derivatives

[ ]:
```
demo1()
```

```
eps=0.01  : f'(2.0) = approx. 12.060099999999707
eps=0.001 : f'(2.0) = approx. 12.006000999997823
eps=0.0001: f'(2.0) = approx. 12.000600010022566
eps=1e-05 : f'(2.0) = approx. 12.000060000261213
eps=1e-06 : f'(2.0) = approx. 12.000006002210739
eps=1e-07 : f'(2.0) = approx. 12.000000584322379
eps=1e-08 : f'(2.0) = approx. 11.999999927070348
eps=1e-09 : f'(2.0) = approx. 12.000000992884452
eps=1e-10 : f'(2.0) = approx. 12.000000992884452
```

[ ]:
```
demo1(derivative=derivative_v1)
```

```
eps=0.01  : f'(2.0) = approx. 12.000099999999847
eps=0.001 : f'(2.0) = approx. 12.000000999998317
eps=0.0001: f'(2.0) = approx. 12.00000001000845
eps=1e-05 : f'(2.0) = approx. 12.00000000021184
eps=1e-06 : f'(2.0) = approx. 12.000000000789157
eps=1e-07 : f'(2.0) = approx. 11.99999999368373
eps=1e-08 : f'(2.0) = approx. 11.999999882661427
eps=1e-09 : f'(2.0) = approx. 12.000000992884452
eps=1e-10 : f'(2.0) = approx. 12.000000992884452
```

For normal-differencing, "if function-values and derivatives are roughly on the scale of 1", we get 'best approximation' for eps ~ 1e-8, which corresponds to about half the available 15-digit accuracy.

With smaller step sizes, we get more valid digits for the change-in-value, but representing the step-size itself numerically vs. the change-in-value becomes more inaccurate, and the best trade-off is "if step size splits the available accuracy in the middle".

For symmetric-differencing, this is a bit trickier to analyze, but as we would expect, "larger step size gives us still good values for higher order approximations."

The way to reason this out is as follows: symmetric-differencing actually corresponds to how we would obtain the linear term when fitting a Taylor polynomial to 2nd order, evaluating the function at: (-eps, 0, +eps). For the linear term, the weight of f(x0) just happens to be zero. If values are "on the scale of 1", then a 3rd-order output-noise of 1e-15 is created by an input-noise of ~1e-5.

7.1 Question (Trick question?): What is the best step size for `derivative_v0`?

[ ]:
```
# Explore yourself!
my_function = lambda x: x**3  # Or feel free to define any other function you␣
↪like.
derivative_v0(my_function, 1.0, eps=0.01)
```

[ ]: 3.0301000000000133

7.1.1 Answer (... which will lead us to FM-AD and RM-AD.)

[ ]:
```
f = lambda x: x**3
print(derivative_v1(f, 1.0, eps=1e-15j))
# Also:
print(derivative_v1(f, 1.0, eps=1e-100j))
```

```
(3+0j)
(3+0j)
```

This may "look like cheating". What is going on here?

- Quite a few Python functions are happy to take complex arguments.
- (Python syntax for complex numbers: `{float_literal}[+-]{float_literal}j`.

  Also: `complex(2, 3) => 2+3j`; `j` as in electrical engineering, where `i` generally is current-density.)
- Both for real and imaginary part, we have 64-bit floating point accuracy.
- We are basically doing the computation for `x0+{fuzz}`.
- Making the "fuzz" imaginary, the imaginary part of the quantities we compute tells us by how much the function's value changes as multiples of the imaginary fuzz, if we fuzz the input like this.
- Note that `(k * 1e-15j)**2 = -(k**2) * 1e-30`.

  So if {function value} "is on the scale of 1", and we can represent numbers "to 15 digits behind the point (relative to that scale)", as long as this derivative k will be smaller in magnitude than 1e15, we get a "numerically accurate" result (= the derivative is as good as it gets with numerics, there are no higher order distortions).
- (Feel free to try this out with a function with larger derivative, such as `(1 + 100*x)**4` at `x=1`.)
- We sort-of abused complex numbers (and holomorphicity) here. But suppose we had a data type `complex0` that differed from the complex numbers by obeying $I^2 = 0$ rather than $i^2 = -1$. These are the called "dual numbers". With those, we could just evaluate `f(x0+I)` and pick out the "imaginary part" to get the derivative. No need for this "1e-15" scaling.
- Suppose now we had some $\mathbb{R} \to \mathbb{R}^{1000}$ function f. Using "dual numbers", we can easily compute $f'(x_0)$ alongside the computation of $f(x_0)$ "with little extra effort". ("little" == "bounded by a small constant effort factor vs. original computation".) Doing this with a complex-numbers-like numeric data type is called "forward mode automatic differentiation (FMAD)".
- We of course also could do this "Forward Mode AD" for 2, 3, etc. parameters, using an algebra of numbers `a+bI+cJ+dK+eL` where we have $I^2 = IJ = IK = \ldots = K^2 = KL = \ldots = L^2 = 0$ ("all 2nd order quantities can be dropped").

  Having n such "infinitesimals" basically multiplies the effort by n.

- (In undergraduate physics, we typically encounter this idea in the form of "Gaussian error propagation".)
- For *numerical optimization*, we have the opposite situation: *one* objective, but "thousands of parameters": $\mathbb{R}^{1000} \to \mathbb{R}$, not $\mathbb{R} \to \mathbb{R}^{1000}$.
- There is a "dual trick" that makes this other case fast. This is known as "reverse mode automatic differentiation" ("RMAD", or just "AD"), or also "sensitivity backpropagation".
- RMAD was first fully formalized as "this is an algorithm that transforms an algorithm computing f into an efficient algorithm computing f'" in Bert Speelpenning's 1980 PhD thesis [15]. Unaware of progress in numerical optimization / engineering, it was rediscovered in the much narrower context of neural networks (so, not as a generic algorithm-transformation) in 1985, in a paper by Rummelhart, Hinton, and Williams [13]. Given that this was rediscovered many times in different contexts, it is hard to name an inventor. Seppo Linnainmaa might have been the first to put this onto computers in the 1970s, but one can demonstrate that the idea is very closely related to insights that go back to Lev Pontryagin, and even William Rowan Hamilton.

8 Before we move on... a Python interlude.

Here is a function that maps some $\mathbb{R} \mapsto \mathbb{R}$ function `f` to a function numerically computing `f'` by finite-differencing:

[ ]:
```
def numerical_derivative(f, default_eps=1e-7):
  """Maps a R->R function to its finite-differencing derivative-function."""
  def fprime(x0, eps=None):
    """Numerically computes the derivative."""
    # If the optional parameter 'eps' was provided, we use that as step size,
    # otherwise we use the value of `default_eps` from the (lexically) outer
    # function as it was at the point when code flow encountered the definition
    # of this inner function.
    epsilon = default_eps if eps is None else eps
    return (f(x0 + epsilon) - f(x0 - epsilon)) / (2 * epsilon)
  return fprime  # We return the derivative-function as a value!
```

Examples below. (Feel free to try out your own!)

[ ]:
```
f1 = lambda x: math.sin(x)
f2 = lambda x: (1-x*x)**.5

f1prime = numerical_derivative(f1)
f2prime = numerical_derivative(f2, default_eps=1e-4)

print(f1prime(math.pi/4))
print(f1prime(math.pi/4, eps=1e-7))

print(f2prime(0.5))  # Uses the default eps-value 1e-4.
print(f2prime(0.5, eps=1e-7))
```

```
0.7071067803510189
0.7071067803510189
-0.5773502743211534
-0.5773502692596466
```

This is a beefed-up variant that uses some Python tricks we have not discussed yet, but is practically useful for symmetric-differencing a $K^{n_1 \times n_2 \times \cdots} \to K^{m_1 \times m_2 \times \cdots}$ tensor-valued function - for $K = \mathbb{R}$ or $K = \mathbb{C}$ - of coordinates that in themselves may be tensor-valued:

[ ]:
```
def n_grad(f, eps=1e-8):
  """Maps a tensor-valued function of tensor-coordinates to a gradient-function.
 ↪"""
  def grad_f(xs):
    xs = numpy.asarray(xs)
    xs0 = xs.ravel()
    dim = xs.size
    xs1 = xs0.copy()
    f0 = numpy.asarray(f(xs0.reshape(xs.shape)))
    result = numpy.zeros(f0.shape + (dim,))
    for n in range(dim):
      xs1[n] = xs0[n] + eps
      fplus = numpy.asarray(f(xs1.reshape(xs.shape)))
      xs1[n] = xs0[n] - eps
      fminus = numpy.asarray(f(xs1.reshape(xs.shape)))
      xs1[n] = xs0[n]
      result[..., n] = (fplus - fminus) / (2 * eps)
    return result.reshape(f0.shape + xs.shape)
  return grad_f

# Example: the gradient of matrix-squaring evaluated at the 2x2-identity -
# which is a [2,2,2,2]-tensor.
grad_matrix_squaring = n_grad(lambda m: m @ m)
print(grad_matrix_squaring(numpy.eye(2)).round(3))
```

```
[[[[2. 0.]
   [0. 0.]]

  [[0. 2.]
   [0. 0.]]]


 [[[0. 0.]
   [2. 0.]]

  [[0. 0.]
   [0. 2.]]]]
```

9 Reverse Mode AD, Step-by-Step

We need a simple-but-not-too-simple toy function to discuss the idea behind Reverse-mode AD.

Suppose you have a 4d box (3d would lead to "too simple" an example) with edge lengths a, b, c, d. The hypersurface of this box is given by the following function.

(Note that putting an ϵ-thickness layer of paint onto a 4d such box would require $\mathrm{Vol}(a+\epsilon, b+\epsilon, c+\epsilon, d+\epsilon) - \mathrm{Vol}(a,b,c,d) + \mathcal{O}(\epsilon^2)$ much paint, and dividing by ϵ gives us the surface area, just as "summing the surface 3d-boxes" does.)

[ ]:
```
def box4d_hypersurface(a, b, c, d):
  """Hypersurface of a 4d box with edge lengths a, b, c, d."""
  # We try to maximize sharing of computations, so rather than computing
  # a * b * c + a * b * d, etc., we remember intermediate products.
  ab = a * b
  cd = c * d
  abc = ab * c
  abd = ab * d
  acd = a * cd
  bcd = b * cd
  half_hypersurface = abc + abd + acd + bcd
  hypersurface = 2 * half_hypersurface
  # Normally, we would instead just do:
  #   return 2 * (abc + abd + acd + bcd)
  # ...but doing smaller steps will be useful for the subsequent discussion.
  return hypersurface
```

[ ]:
```
# Merely ensuring that this computes a plausible value at all for some simple␣
↪case.
print(box4d_hypersurface(10, 10, 10, 10))
```

```
8000
```

Starting from the function definition above, let us manually implement a function that "computes the gradient at a given point".

For some given point `(a, b, c, d)`, we want to know `grad(box4d_hypersurface)(a, b, c, d)`, i.e. "by how much does the hypersurface change if we tweak `a` by ϵ (relative to ϵ), and likewise for `b`, `c`, `d`.

People did this sort of thing (manually implementing gradients) for quite a while, and gradually realized a few tricks that condensed into this generally-applicable recipe. Here: for "straight" functions without loops (as this all was understood around 1976 already) - and subsequently, for code with branches and loops and other control structure.

The Recipe

1. Copy the code that computes the function. Efficiently computing the gradient will start with computing the forward-function.

Do make sure the function is written in such a way that intermediate results never are overwritten.

(This already holds for our example.)

2. Comment out the final 'return ...' statement.

   Rather than returning that value, we will have to do more work afterwards.

3. There will (in general) be quite a few intermediate quantities for this calculation. For the k-th such intermediate quantity - q_k, we introduce a "sensitivity accumulator" S_{q_k}, which initially is initialized to zero right at the point where q_k is introduced.

4. We also treat the inputs and the output as "intermediate quantities".

5. Right from the point where we could return the result, go through the calculation once again, but this time backwards, step-by-step, from each intermediate quantity back to earlier ones in terms of which it is defined. While doing so, maintain this invariant:

   Each such "sensitivity accumulator" S_{q_k} keeps track of "by how much would the final result of this function change, relative to ϵ, if I intercepted code flow right after this line in the calculation that uses the quantity q_k, and made an ad-hoc change to q_k, incrementing it by ϵ?

Once we know the sensitivities of the final result on the input parameters, we have our gradient.

Let us apply this recipe step-by-step.

[ ]:
```python
# Step 1: Copying the code.

def box4d_hypersurface_step1(a, b, c, d):
  """Hypersurface of a 4d box with edge lengths a, b, c, d."""
  # We try to maximize sharing of computations, so rather than computing
  # a * b * c + a * b * d, etc., we remember intermediate products.
  ab = a * b
  cd = c * d
  abc = ab * c
  abd = ab * d
  acd = a * cd
  bcd = b * cd
  half_hypersurface = abc + abd + acd + bcd
  hypersurface = 2 * half_hypersurface
  # Normally, we would instead just do:
  #   return 2 * (abc + abd + acd + bcd)
  # ...but doing smaller steps will be useful for the subsequent discussion.
  return hypersurface


# Step 2: Removing the `return ...`

def box4d_hypersurface_step2(a, b, c, d):
  """Hypersurface of a 4d box with edge lengths a, b, c, d."""
  # We try to maximize sharing of computations, so rather than computing
```

```
  # a * b * c + a * b * d, etc., we remember intermediate products.
  ab = a * b
  cd = c * d
  abc = ab * c
  abd = ab * d
  acd = a * cd
  bcd = b * cd
  half_hypersurface = abc + abd + acd + bcd
  hypersurface = 2 * half_hypersurface
  # Normally, we would instead just do:
  #   return 2 * (abc + abd + acd + bcd)
  # ...but doing smaller steps will be useful for the subsequent discussion.
  ### Rather than returning the value, proceed to compute the gradient.
  ### return hypersurface


# Step 3: Introduce sensitivity accumulators.
#
# Note: Using Semicolons is generally frowned-upon-style in Python,
# but here it comes handy for didactic purposes.
# Still: avoid this "in real code".

def box4d_hypersurface_step3(a, b, c, d):
  """Hypersurface of a 4d box with edge lengths a, b, c, d."""
  # We try to maximize sharing of computations, so rather than computing
  # a * b * c + a * b * d, etc., we remember intermediate products.
  ab = a * b; s_ab = 0
  cd = c * d; s_cd = 0
  abc = ab * c; s_abc = 0
  abd = ab * d; s_abd = 0
  acd = a * cd; s_acd = 0
  bcd = b * cd; s_bcd = 0
  half_hypersurface = abc + abd + acd + bcd; s_half_hypersurface = 0
  hypersurface = 2 * half_hypersurface
  # Normally, we would instead just do:
  #   return 2 * (abc + abd + acd + bcd)
  # ...but doing smaller steps will be useful for the subsequent discussion.
  ### Rather than returning the value, proceed to compute the gradient.
  ### return hypersurface

# Step 4: Also do this for the inputs and for the result value.

def box4d_hypersurface_step4(a, b, c, d):
  """Hypersurface of a 4d box with edge lengths a, b, c, d."""
  # We try to maximize sharing of computations, so rather than computing
  # a * b * c + a * b * d, etc., we remember intermediate products.
  s_a = 0; s_b = 0; s_c = 0; s_d = 0    # <== !!!
```

```
  ab = a * b; s_ab = 0
  cd = c * d; s_cd = 0
  abc = ab * c; s_abc = 0
  abd = ab * d; s_abd = 0
  acd = a * cd; s_acd = 0
  bcd = b * cd; s_bcd = 0
  half_hypersurface = abc + abd + acd + bcd; s_half_hypersurface = 0
  ### Rather than returning the value, proceed to compute the gradient.
  ### return hypersurface
  # !!! `hypersurface` is our result value. We already know the final
  # !!!  value of `s_hypersurface`: If we intercepted the calculation after
  # !!! `hypersurface = 2 * ...`, and added a change-forcing line...:
  # !!!    hypersurface = hypersurface + epsilon
  # !!!  this would (trivially) change the function's result by epsilon.
  # !!!  So, s_hypersurface = 1.
  hypersurface = 2 * half_hypersurface; s_hypersurface = 1
  # Normally, we would instead just do:
  #   return 2 * (abc + abd + acd + bcd)
  # ...but doing smaller steps will be useful for the subsequent discussion.

# Step 5 ("backpropagation") - we do this one-by-one.

def box4d_hypersurface_step5a(a, b, c, d):
  """Hypersurface of a 4d box with edge lengths a, b, c, d."""
  # We try to maximize sharing of computations, so rather than computing
  # a * b * c + a * b * d, etc., we remember intermediate products.
  s_a = 0; s_b = 0; s_c = 0; s_d = 0   # <== !!!
  ab = a * b; s_ab = 0
  cd = c * d; s_cd = 0
  abc = ab * c; s_abc = 0
  abd = ab * d; s_abd = 0
  acd = a * cd; s_acd = 0
  bcd = b * cd; s_bcd = 0
  half_hypersurface = abc + abd + acd + bcd; s_half_hypersurface = 0
  hypersurface = 2 * half_hypersurface; s_hypersurface = 1
  ### Rather than returning the value, proceed to compute the gradient.
  ### return hypersurface
  # Forward-computation:
  #    hypersurface = 2 * half_hypersurface
  # If we had added a line right before this that reads:
  #    half_hypersurface = half_hypersurface + eps
  # ...this would change `hypersurface` by 2 * eps, and the end result by
  # 2 * eps * {sensitivity of the end result on `hypersurface`}
  # (this is the chain rule). So, `s_half_hypersurface` receives a contribution:
  s_half_hypersurface = s_half_hypersurface + 2 * s_hypersurface
  # (From now on, we will just write this:
  #    s_half_hypersurface += 2 * s_hypersurface
```

```
  #  ...even though one should in general stay away from the += operator
  #  in Python, at least while learning the language.)

# Then...

def box4d_hypersurface_step5b(a, b, c, d):
  """Hypersurface of a 4d box with edge lengths a, b, c, d."""
  # We try to maximize sharing of computations, so rather than computing
  # a * b * c + a * b * d, etc., we remember intermediate products.
  s_a = 0; s_b = 0; s_c = 0; s_d = 0   # <== !!!
  ab = a * b; s_ab = 0
  cd = c * d; s_cd = 0
  abc = ab * c; s_abc = 0
  abd = ab * d; s_abd = 0
  acd = a * cd; s_acd = 0
  bcd = b * cd; s_bcd = 0
  half_hypersurface = abc + abd + acd + bcd; s_half_hypersurface = 0
  hypersurface = 2 * half_hypersurface; s_hypersurface = 1
  ### Rather than returning the value, proceed to compute the gradient.
  ### return hypersurface
  # Forward-computation:
  #    hypersurface = 2 * half_hypersurface
  s_half_hypersurface += 2 * s_hypersurface
  # Forward-computation:
  #    half_hypersurface = abc + abd + acd + bcd
  s_abc += s_half_hypersurface
  s_abd += s_half_hypersurface
  s_acd += s_half_hypersurface
  s_bcd += s_half_hypersurface

# Then:

def box4d_hypersurface_step5c(a, b, c, d):
  """Hypersurface of a 4d box with edge lengths a, b, c, d."""
  # We try to maximize sharing of computations, so rather than computing
  # a * b * c + a * b * d, etc., we remember intermediate products.
  s_a = 0; s_b = 0; s_c = 0; s_d = 0   # <== !!!
  ab = a * b; s_ab = 0
  cd = c * d; s_cd = 0
  abc = ab * c; s_abc = 0
  abd = ab * d; s_abd = 0
  acd = a * cd; s_acd = 0
  bcd = b * cd; s_bcd = 0
  half_hypersurface = abc + abd + acd + bcd; s_half_hypersurface = 0
  hypersurface = 2 * half_hypersurface; s_hypersurface = 1
  ### Rather than returning the value, proceed to compute the gradient.
  ### return hypersurface
```

```
  # Forward-computation:
  #    hypersurface = 2 * half_hypersurface
  s_half_hypersurface += 2 * s_hypersurface
  # Forward-computation:
  #    half_hypersurface = abc + abd + acd + bcd
  s_abc += s_half_hypersurface
  s_abd += s_half_hypersurface
  s_acd += s_half_hypersurface
  s_bcd += s_half_hypersurface
  # Forward-computation:
  #     bcd = b * cd
  # For each of the factors, sensitivity = {sensitivity of the
  # end result on the left hand side intermediate quantity, which we at
  # this point fully know, since we reached the definition of this quantity,
  # and all its uses come after the definition} * {other-factor, which scales
 ↪it}.
  # (This is just the product rule in action).
  s_b += s_bcd * cd
  s_cd += b * s_bcd

# Proceeding further like this, now doing larger steps -
# and completing the work.

def box4d_hypersurface_and_gradient(a, b, c, d):
  """Hypersurface of a 4d box with edge lengths a, b, c, d."""
  # We try to maximize sharing of computations, so rather than computing
  # a * b * c + a * b * d, etc., we remember intermediate products.
  s_a = 0; s_b = 0; s_c = 0; s_d = 0   # <== !!!
  ab = a * b; s_ab = 0
  cd = c * d; s_cd = 0
  abc = ab * c; s_abc = 0
  abd = ab * d; s_abd = 0
  acd = a * cd; s_acd = 0
  bcd = b * cd; s_bcd = 0
  half_hypersurface = abc + abd + acd + bcd; s_half_hypersurface = 0
  hypersurface = 2 * half_hypersurface; s_hypersurface = 1
  ### Rather than returning the value, proceed to compute the gradient.
  ### return hypersurface
  # Forward-computation:
  #    hypersurface = 2 * half_hypersurface
  s_half_hypersurface += 2 * s_hypersurface
  # Forward-computation:
  #    half_hypersurface = abc + abd + acd + bcd
  s_abc += s_half_hypersurface
  s_abd += s_half_hypersurface
  s_acd += s_half_hypersurface
  s_bcd += s_half_hypersurface
```

```
  # Forward-computation:
  #     bcd = b * cd
  s_b += s_bcd * cd
  s_cd += b * s_bcd
  # Forward-computation:
  #     acd = a * cd
  s_a += s_acd * cd
  s_cd += a * s_acd  # Note that this 2nd(-in-reverse-order) use of `cd`
                     # introduces a 2nd contribution to `s_cd`!
  # Forward-Computation:
  #     abd = ab * d
  s_ab += s_abd * d
  s_d += ab * s_abd
  # Forward-computation:
  #     abc = ab * c
  s_ab += s_abc * c
  s_c += ab * s_abc
  # Forward-computation:
  #     cd = c * d
  s_c += s_cd * d
  s_d += c * s_cd
  # Forward-computation:
  #    ab = a * b
  s_a += s_ab * b
  s_b += a * s_ab
  ### At this point, we know the sensitivities of the result on the vector of
  ### input-coordinates. That is just the gradient.
  ### "Since we can", we also return the function's value (as it may be useful):
  return (hypersurface, (s_a, s_b, s_c, s_d))
```

[ ]:
```
# Let's try it out...:

print(box4d_hypersurface_and_gradient(2, 5, 7, 11))
print(derivative_v1((lambda x: box4d_hypersurface(2, x, 7, 11)), 5, eps=1e-15j))
```

```
(1438, (334, 226, 174, 118))
(225.99999999999997+0j)
```

9.1 Computational Complexity

When reasoning about the effort needed to obtain a good gradient, a key observation is that "per use of an intermediate quantity", the effort is a fixed multiple of the corresponding effort done on the forward-calculation.

"The most painful thing that can happen" is back-propagating through a product. We need to zero-initialize two sensitivity-accumulators (somewhere), and to what extent this can be amortized depends e.g. on the operating system and code structure. Then, we need to load these accumulators, add the increments, and store them again. Computing the increments requires a multiplication each. We can write down effort as a function of the relative cost of load/add/multiply/store operations,

and then study the range of the {gradient effort} / {forward pass effort} function as a function of these costs. Speelpenning's thesis gives a bound of "at most 8x effort", but one can reason out "at most 6x", and in practice one often observes "factor 4-5, rarely around 6". However, on real computers, cache performance also is an issue here.

[ ]:
```
# Hand-backpropagation can be a bit tedious, and one needs to be extremely
# careful to avoid typos, but the good news is: If something is off, we can
# always modify the forward-calculation "to allow injecting epsilon changes"
# and then compare "measured" and "computed" sensitivities, zeroing in
# on the location of the problem.
#
# This might roughly look as follows...:


def box4d_hypersurface_perturbed(a, b, c, d, eps_cd=0, eps_abc=0):
  """Hypersurface of a 4d box with edge lengths a, b, c, d."""
  # We try to maximize sharing of computations, so rather than computing
  # a * b * c + a * b * d, etc., we remember intermediate products.
  ab = a * b
  cd = c * d + eps_cd
  abc = ab * c + eps_abc
  abd = ab * d
  acd = a * cd
  bcd = b * cd
  half_hypersurface = abc + abd + acd + bcd
  hypersurface = 2 * half_hypersurface
  # Normally, we would instead just do:
  #   return 2 * (abc + abd + acd + bcd)
  # ...but doing smaller steps will be useful for the subsequent discussion.
  return hypersurface

print((box4d_hypersurface_perturbed(2, 5, 10, 20, eps_cd=1e-7) -
       box4d_hypersurface_perturbed(2, 5, 10, 20)) / 1e-7)
# ...which we then can compare against the finalized s_cd, respectively s_abc.

# We will see a more complicated semi-realistic worked out example below which
# still has some of its gradient debug code deliberately left in.
```

```
13.999997463542968
```

9.2 More complicated code structure

Having seen the basic idea, it is natural to next discuss three generalizing aspects:

1. "data-parallelism".
2. Conditionals
3. Loops

What we have seen so far looks at individual number-additions and multiplications.

However, the reasoning carries over straightaway to tensor arithmetics.

Let's look at an example for backpropagating through some matrix/tensor operation.

For pedagogical reasons (and also to practice using einsum()), we will convert all (multi)linear operations to tensor contractions.

"Once we know how to backprop through einsum(), we understood how to do this for tensor arithmetics."

[ ]:
```
# Backpropagating tensor operations.

G_NEWTON = 6.672e-11 # m^3 kg^(-1) s^(-2)

def gravitational_potential(m1, m2, pos1, pos2):
  v1 = numpy.asarray(pos1, dtype=numpy.float64)
  v2 = numpy.asarray(pos2, dtype=numpy.float64)
  v12 = v2 - v1
  dist_squared = numpy.einsum('i,i->', v12, v12)
  dist = dist_squared**.5
  m12 = m1 * m2
  Gm1m2 = G_NEWTON * m12
  energy = Gm1m2 / dist
  return -energy

# We backpropagate into every parameter here.
def gravitational_potential_and_grad(m1, m2, pos1, pos2):
  s_m1 = 0; s_m2 = 0
  v1 = numpy.asarray(pos1, dtype=numpy.float64) ; s_v1 = numpy.zeros_like(v1)
  v2 = numpy.asarray(pos2, dtype=numpy.float64) ; s_v2 = numpy.zeros_like(v2)
  v12 = v2 - v1; s_v12 = numpy.zeros_like(v12)
  dist_squared = numpy.einsum('i,i->', v12, v12); s_dist_squared = 0
  dist = dist_squared**.5; s_dist = 0
  m12 = m1 * m2 ; s_m12 = 0
  Gm1m2 = G_NEWTON * m12; s_Gm1m2 = 0
  energy = Gm1m2 / dist; s_energy = 0
  # return -energy
  s_energy += -1
  s_Gm1m2 += s_energy / dist
  # Here, we nomimally would have to use "inv_dist" as an intermediate quantity
  # and backprop through f(x)=1/x, but let's shorten this.
  s_dist += s_energy * Gm1m2 * (-1 / dist_squared)
  s_m12 = G_NEWTON * s_Gm1m2
  s_m1 += s_m12 * m2
  s_m2 += m1 * s_m12
  s_dist_squared += s_dist * 0.5 / dist
  # These two could of course be combined into a simple
  # s_v12 += 2 * s_dist_squared * v12
  s_v12 += numpy.einsum(',i->i', s_dist_squared, v12)
```

```
  s_v12 += numpy.einsum('i,->i', v12, s_dist_squared)
  s_v2 += s_v12
  s_v1 -= s_v12  # Note -= !
  return -energy, (s_m1, s_m2, s_v1, s_v2)



pprint.pprint(gravitational_potential_and_grad(2e7, 3e7, [5, 10, 15], [5, 20,␣
↪30]))
def debug_potential_1parameter(x):
  return gravitational_potential(2e7, 3e7, [5, 10, 15 + x], [5, 20, 30])
print('===')
pprint.pprint(derivative_v1(debug_potential_1parameter, 0.0, eps=1e-8))
```

```
(-2220.5758255288374,
 (-0.00011102879127644187,
  -7.401919418429458e-05,
  array([   0.        ,  -68.32541002, -102.48811502]),
  array([  0.        ,  68.32541002, 102.48811502])))
===
-102.48811577184824
```

So, not breaking up tensor arithmetics into individual additions and multiplications and then doing backpropagation on this makes less sense than keeping the tensor arithmetics intact.

Next, let us look into some nontrivial control structure. We will use Heron's method for root finding by iteratively averaging {candidate-root} and {square}/{candidate-root}.

[ ]:
```
def heron_root(x, max_distance=1e-7):
  """Computes sqrt(x) by iterative averaging."""
  y = 1
  while True:
    x_div_y = x / y
    mid = 0.5 * (y + x_div_y)
    if abs(x_div_y - y) <= max_distance:
      return mid
    else:
      y = mid

print(heron_root(2))
```

```
1.414213562373095
```

Our first task is to convert this simple function to a form that "remembers all intermediate quantities, and never overwrites anything".

For the sake of clarity, this form deliberately ignores some optimization opportunities. Rather, we emphasize "understandability" here (and also get to see some useful new Python).

[ ]:
```python
import dataclasses

# Nominally, we are introducing a class here,
# but there is no actual state-management: The `HeronState`
# instance is effectively just a fancy tuple with named slots.
@dataclasses.dataclass(frozen=True)
class HeronState:
  y : float
  x_div_y : float
  mid : float
  delta : float

def heron_root_v1(x, max_distance=1e-7):
  """Computes sqrt(x) by iterative averaging."""
  states = []
  y = 1
  while True:
    x_div_y = x / y
    mid = 0.5 * (y + x_div_y)
    delta = x_div_y - y
    if abs(delta) <= max_distance:
      return mid
    else:
      y = mid
```

Next, we can think about backpropagating this.

[ ]:
```python
def heron_root_and_grad(x, max_distance=1e-7):
  """Computes sqrt(x) and by iterative averaging - plus its gradient."""
  states = []
  y = 1
  while True:
    x_div_y = x / y
    mid = 0.5 * (y + x_div_y)
    delta = x_div_y - y
    states.append(HeronState(y=y, x_div_y=x_div_y, mid=mid, delta=delta))
    if abs(delta) <= max_distance:
      # Rather than returning to the caller, we need to exit to after-the-loop.
      # return mid
      break
    else:
      y = mid
  # At this point,
  # - The result we would have returned is in `mid`.
  # - List entry `states[n]` contains intermediate quantites right at the end
  #   of iteration n, before we updated `y`.
  result = mid
  s_x = 0
```

```
  s_mid = 1
  for state in reversed(states):
    s_y = 0.5 * s_mid
    s_x_div_y = 0.5 * s_mid
    s_x += s_x_div_y / state.y
    s_y += s_x_div_y * x * (-1 / (state.y * state.y))
    # At the end of the previous loop, s_y was assigned from `mid`.
    s_mid = s_y
  # Here, it turns out that we actually would not have had to remember `delta`,
  # since this was not used as an intermediate quantity, but only to make
  # some continue-or-not decisions that became retraceable via the bookkeeping
  # we do with `states`.
  # Let us return some more detailed information.
  return dict(result=result,
              num_iterations=len(states),
              grad=s_x)


print(heron_root_and_grad(100.0))
print('###')
# Given that this method is reasonably efficient, we have to pick really large
# max_distance values to make it "use too few iterations".
# The question here is: Do these cases then produce good gradients?
print(heron_root_and_grad(100.0, max_distance=10))
print(derivative_v1(
    (lambda x: heron_root(x, max_distance=10)),
    100.0,
    eps=1e-7))
print('###')
print(heron_root_and_grad(100.0, max_distance=50))
print(derivative_v1(
    (lambda x: heron_root(x, max_distance=50)),
    100.0,
    eps=1e-7))
```

```
{'result': 10.0, 'num_iterations': 8, 'grad': 0.05}
###
{'result': 10.840434673026925, 'num_iterations': 4, 'grad': 0.06835572803193413}
0.06835572108343513
###
{'result': 26.24009900990099, 'num_iterations': 2, 'grad': 0.2500980296049407}
0.25009802229192246
```

So, backpropagating through loops is generally a bit more messy, but actually doable - even if the number of iterations is data-dependent.

The basic principle is: When computing the gradient, as we are retracing the computational steps, we make every computational flow decision (which branch to take, when to leave a loop) as on the forward computation.

10 Actually doing optimization

Now that we have a way to efficiently get good gradients of even high-dimensional objective functions, let us actually explore what we can do with this. Numerical optimization obviously is an important topic in itself.

For the time being, let us focus on simple non-ML problems to practice gradient-based numerical optimization with `<=1000` parameters.

The short story here is: For nonlinear minimization, if the function can be approximated by some quadric near the minimum in some reasonable sense, and if it does not have too weird 'hole on the golf course' minima, the BFGS-family of algorithms (Broyden-Fletcher-Goldfarb-Shanno) generally are a very potent tool one should know about. The most general one that is available in the `scipy.optimize` module is L-BFGS-B.

The basic idea is: We compute gradients to find out in which direction to move in order to minimize the function. From successive gradient evaluations, we estimate a (`NxN`) approximate Hessian and use this to turn the gradient (a "covector"!) into a step(-vector).

Let us consider the following problem: We have two wire circles that extend in x-y direction, have their centers at `(0, 0, z1)`, respectively `(0, 0, z2)`, with radii `r1` and `r2`, and want to find the shape of some soap film that stretches between these. There are two ways to see the physics:

1. "Local law": Mean (extrinsic) curvature ~ pressure difference on both sides of the soap film = 0.
2. "Variational principle": Total surface tension energy ~ surface area, is minimal.

Let us use the latter principle - and also make some basic assumptions here:

1. We can get a good approximation to the geometry by finding area-minimizing vertices of a triangle mesh.
2. We can simplify this further by meshing a conical frustum and allowing every vertex a single degree of freedom - its distance from the symmetry axis.

(Via 2., we "fix gauge symmetry" related to being able to move the discretization points along the surface.)

Here, we will go for a slightly unusual approach and compute triangle areas from side-lengths, using (again) a formula by Heron.

(Admittedly, this first example is slightly silly, as we are not exploiting the symmetry of the problem. The first exercise will break that symmetry.)

Participant Exercise: Replace Heron's formula with code that computes triangle area via the cross product of two sides (and also backpropagate this to obtain its gradient).

[ ]:
```
def heron_area(d1, d2, d3):
  """Computes the area of a triangle with sides `d1, d2, d3`. Vectorized."""
  s = (d1 + d2 + d3) * 0.5
  return numpy.sqrt(s * (s - d1) * (s - d2) * (s - d3))


def vertices_and_total_mesh_area(z1, z2, r1, r2,
```

```
                                inner_vertex_radial_deviations,
                                # This is debugging code deliberately left
                                # in this example for illustration - how to
                                # track down gradient discrepancies.
                                ddd_delta_xyzs_wrapped=(),
                                ):
  """Computes vertex-positions and total mesh area.

  Args:
    z1: float, z-height of the first ring.
    z2: float, z-height of the 2nd ring.
    r1: float, radius of the first ring.
    r2: float, radius of the 2nd ring.
    inner_vertex_radial_deviations:
      float [num_inner_rings, num_vertices_per-ring] numpy.ndarray-like,
      per inner mesh-vertex, the (signed) distance by which the vertex
      has been moved radially out from the x=y=0 symmetry axis relative to
      a point on the surface of the conical frustum that is the convex hull
      of the circles around x=y=0 specified by (z1, r1) and (z2, r2).

  Returns:
    A pair `(vertex_3d_coords, total_area)`, where `vertex_3d_coords` is a
    float [num_rings_total, num_vertices_per_ring + 1, 3]-numpy.ndarray with
    vertex coordinates (including a wraparound-column),
    and `total_area` is the total surface area.
  """
  inner_vertex_radial_deviations = numpy.asarray(inner_vertex_radial_deviations)
  num_inner_rings, num_vertices_per_ring = inner_vertex_radial_deviations.shape
  num_rings = 2 + num_inner_rings  # The fixed 'wire rings'.
  rs = numpy.linspace(r1, r2, num_rings)
  # Per-vertex z-coordinates.
  zs = numpy.tile(numpy.linspace(z1, z2, num_rings)[:, numpy.newaxis],
                  [1, num_vertices_per_ring])
  vertex_radial_deviations = numpy.pad(
      inner_vertex_radial_deviations, [(1, 1), (0, 0)])
  xys_unscaled_complex = numpy.exp(
      1j * numpy.linspace(0, 2 * numpy.pi, num_vertices_per_ring + 1))[:-1]
  xys_scaled_complex = xys_unscaled_complex[numpy.newaxis, :] * (
      rs[:, numpy.newaxis] + vertex_radial_deviations)
  xyzs = (  # shape [num_rings, num_vertices_per_ring, 3]
      numpy.stack([xys_scaled_complex.real,
                   xys_scaled_complex.imag,
                   zs],
                  axis=-1))
  xyzs_wrapped = numpy.concatenate([xyzs, xyzs[:, :1, :]], axis=1)
  for delta, coords in ddd_delta_xyzs_wrapped:
    xyzs_wrapped[coords] += delta
```

```
  # There is a 1-to-1 mapping between every point not on the z2-ring and
  # non-planar 4-plaquettes made of two triangles.
  # The four relevant points are:
  # "here" == "SW",
  # "up" (as a shorthand for "towards z2") == "NW",
  # "ccw" (as a shorthand for "next-w.r.t. increasing angle") == "SE",
  # and "ccw-up" == "NE".
  p_sw = xyzs_wrapped[:-1, :-1, :]
  p_nw = xyzs_wrapped[1:, :-1, :]
  p_se = xyzs_wrapped[:-1, 1:, :]
  p_ne = xyzs_wrapped[1:, 1:, :]
  d_sw_nw = numpy.linalg.norm(p_sw - p_nw, axis=-1)
  d_sw_se = numpy.linalg.norm(p_sw - p_se, axis=-1)
  d_se_ne = numpy.linalg.norm(p_se - p_ne, axis=-1)
  d_ne_nw = numpy.linalg.norm(p_ne - p_nw, axis=-1)
  d_se_nw = numpy.linalg.norm(p_se - p_nw, axis=-1)
  area_sw_se_nw = heron_area(d_sw_se, d_se_nw, d_sw_nw)
  area_se_ne_nw = heron_area(d_se_ne, d_ne_nw, d_se_nw)
  return xyzs_wrapped, area_sw_se_nw.sum() + area_se_ne_nw.sum()


# We use the BFGS algorithm, as provided by scipy.optimize, but
# we let scipy determine gradients numerically, but we make the
# gradient-function an optional parameter to this function:
# If it is not provided, BFGS will use finite differencing to compute
# the gradient.
def get_minimal_area(z1, z2, r1, r2,
                     num_inner_rings,
                     num_vertices_per_ring,
                     grad_area=None,
                     report_every_n_calls=1000):
  t0 = time.time()
  t_prev = time.time()
  num_calls = 0
  def f_area(params, also_return_vertices=False):
    nonlocal t_prev, num_calls
    num_calls += 1
    vertex_radial_deviations = numpy.asarray(params).reshape(
        num_inner_rings, num_vertices_per_ring)
    vertices_wrapped, area = vertices_and_total_mesh_area(
        z1, z2, r1, r2, vertex_radial_deviations)
    if num_calls % report_every_n_calls == 0:
      t_now = time.time()
      print(f'[T={t_now - t0:8.3f} s, '
            f'+{t_now-t_prev:8.3f} s for {report_every_n_calls} calls]: '
            f'a={area:16.10f}')
      t_prev = t_now
```

```
    if also_return_vertices:
      # This is slightly messy design: Depending on parameters, we either
      # return only the area (as we need it for optimization), or also
      # the other useful information, the vertices.
      return area, vertices_wrapped
    else:
      return area
  def fprime_area(params):
    # Adapter function - wraps up the gradient-function `grad_area`,
    # which has the same input parameters as `vertices_and_total_mesh_area`
    # to take the same inputs as f_area.
    vertex_radial_deviations = numpy.asarray(params).reshape(
        num_inner_rings, num_vertices_per_ring)
    s_vertex_radial_deviations = grad_area(
        z1, z2, r1, r2, vertex_radial_deviations)
    # Here, we are only interested in the dependency on the radial deviations.
    return s_vertex_radial_deviations.ravel()
  opt_pos = scipy.optimize.fmin_bfgs(
      f_area,
      numpy.zeros(num_inner_rings * num_vertices_per_ring),
      fprime=None if grad_area is None else fprime_area)
  opt_val, vertices_wrapped = f_area(opt_pos, also_return_vertices=True)
  return opt_val, vertices_wrapped
```

[ ]:
```
# Running a demo and plotting the minimal surface.

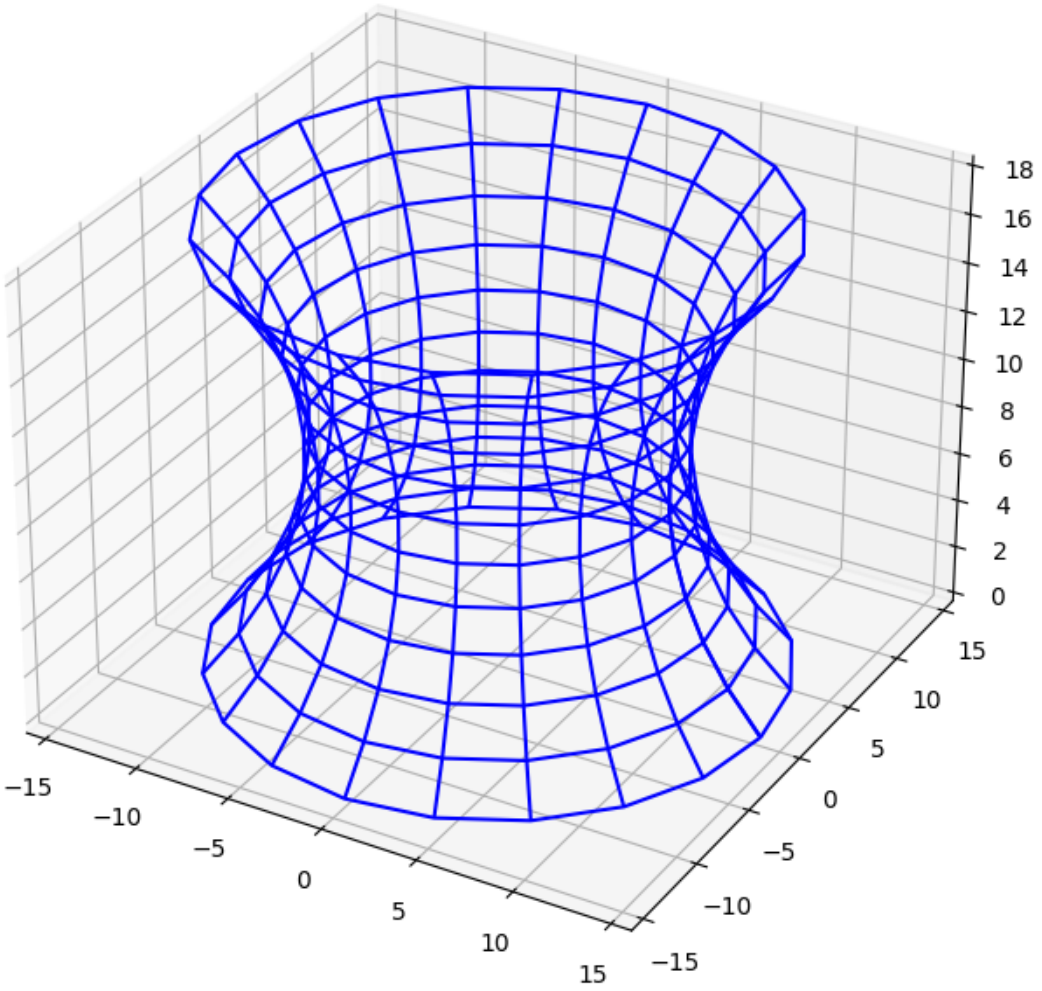
demo_opt_val, demo_vertices = get_minimal_area(0.0, 18, 14.0, 14.0, 10, 20)


pyplot_axes = pyplot.axes(projection='3d')
pyplot_axes.plot_wireframe(demo_vertices[..., 0],
                           demo_vertices[..., 1],
                           demo_vertices[..., 2],
                           color='blue')
```

```
[T=   0.558 s, +   0.558 s for 1000 calls]: a= 1465.9422784433
[T=   1.239 s, +   0.681 s for 1000 calls]: a= 1437.2688034468
[T=   1.844 s, +   0.605 s for 1000 calls]: a= 1437.1263986860
[T=   2.447 s, +   0.603 s for 1000 calls]: a= 1437.1156192768
[T=   3.016 s, +   0.570 s for 1000 calls]: a= 1437.1141902485
[T=   3.590 s, +   0.574 s for 1000 calls]: a= 1437.1141420298
[T=   4.164 s, +   0.574 s for 1000 calls]: a= 1437.1141385439
[T=   4.748 s, +   0.583 s for 1000 calls]: a= 1437.1141383110
[T=   5.265 s, +   0.517 s for 1000 calls]: a= 1437.1141383041
Warning: Desired error not necessarily achieved due to precision loss.
         Current function value: 1437.114138
         Iterations: 32
         Function evaluations: 9648
```

```
            Gradient evaluations: 48
```

[ ]:
```
<mpl_toolkits.mplot3d.art3d.Line3DCollection at 0x7946300fea40>
```



10.1 Let us improve this by adding fast gradients.

This notebook already has the code for the main body, but what is missing (exercise!) is the gradient of Heron's triangle-area formula.

[ ]:
```
def grad_heron_area(d1, d2, d3, s_output):
  """Computes the gradient for `heron_area`."""
  # TODO: Document this properly!
  # s_output is the sensitivity of the final result of the computation
  # where this occurs as an intermediate result on the output of this function.
```

```
  # s = (d1 + d2 + d3) * 0.5
  # return numpy.sqrt(s * (s - d1) * (s - d2) * (s - d3))
  raise NotImplementedError('TODO: Implement this! (Participant Exercise)')

### Solution


def grad_total_mesh_area(z1, z2, r1, r2,
                         inner_vertex_radial_deviations,
                         ddd_delta_xyzs_wrapped=(),
                         ):
  """(Partial) gradient for the `total_mesh_area` function's area.

  Args:
    z1: float, z-height of the first ring.
    z2: float, z-height of the 2nd ring.
    r1: float, radius of the first ring.
    r2: float, radius of the 2nd ring.
    inner_vertex_radial_deviations:
      float [num_inner_rings, num_vertices_per-ring] numpy.ndarray-like,
      per inner mesh-vertex, the (signed) distance by which the vertex
      has been moved radially out from the x=y=0 symmetry axis relative to
      a point on the surface of the conical frustum that is the convex hull
      of the circles around x=y=0 specified by (z1, r1) and (z2, r2).

  Returns:
    float [num_inner_rings, num_vertices_per-ring] numpy.ndarray,
    sensitivities of the mesh area on `inner_vertex_radial_deviations`.
  """
  # In this example, we only use the sensitivities on the radial deviations,
  # but for the sake of illustration, we compute the full gradient.
  #
  # Also, for the sake of "sticking to the recipe", this code does not do
  # some obvious simplifications.
  # When reading this code, it might make sense to duplicate the browser tab
  # and use both tabs side-by-side to view forward and reverse code in
  # separate windows.
  #
  # Participant Exercise: Simplify the code further
  # (such as by eliminating s_area_se_ne_nw, etc.).
  z_like = numpy.zeros_like  # Abbreviation
  inner_vertex_radial_deviations = numpy.asarray(inner_vertex_radial_deviations)
  num_inner_rings, num_vertices_per_ring = inner_vertex_radial_deviations.shape
  num_rings = 2 + num_inner_rings  # The fixed 'wire rings'.
  rs = numpy.linspace(r1, r2, num_rings)
  s_rs = z_like(rs)
  # Per-vertex z-coordinates.
```

```
zs = numpy.tile(numpy.linspace(z1, z2, num_rings)[:, numpy.newaxis],
                [1, num_vertices_per_ring])
s_zs = z_like(zs)
vertex_radial_deviations = numpy.pad(
    inner_vertex_radial_deviations, [(1, 1), (0, 0)])
s_vertex_radial_deviations = z_like(vertex_radial_deviations)
# This is just a constant that only depends on the discrete parameter
# `num_vertices_per_ring` - no need to keep track of a sensitivity here.
xys_unscaled_complex = numpy.exp(
    1j * numpy.linspace(0, 2 * numpy.pi, num_vertices_per_ring + 1))[:-1]
xys_scaled_complex = xys_unscaled_complex[numpy.newaxis, :] * (
    rs[:, numpy.newaxis] + vertex_radial_deviations)
s_xys_scaled_complex = z_like(xys_scaled_complex)
xyzs = (  # shape [num_rings, num_vertices_per_ring, 3]
    numpy.stack([xys_scaled_complex.real,
                 xys_scaled_complex.imag,
                 zs],
                axis=-1))
s_xyzs = z_like(xyzs)
xyzs_wrapped = numpy.concatenate([xyzs, xyzs[:, :1, :]], axis=1)
s_xyzs_wrapped = z_like(xyzs_wrapped)
# There is a 1-to-1 mapping between every point not on the z2-ring and
# non-planar 4-plaquettes made of two triangles.
# The four relevant points are:
# "here" == "SW",
# "up" (as a shorthand for "towards z2") == "NW",
# "ccw" (as a shorthand for "next-w.r.t. increasing angle") == "SE",
# and "ccw-up" == "NE".
p_sw = xyzs_wrapped[:-1, :-1, :]; s_p_sw = z_like(p_sw)
p_nw = xyzs_wrapped[1:, :-1, :]; s_p_nw = z_like(p_nw)
p_se = xyzs_wrapped[:-1, 1:, :]; s_p_se = z_like(p_se)
p_ne = xyzs_wrapped[1:, 1:, :]; s_p_ne = z_like(p_ne)
# Here, we introduce extra intermediate quantities which become the
# arguments of numpy.linalg.norm() below.
# "We want to remember every intermediate quantity."
p_sw_nw = p_sw - p_nw; s_p_sw_nw = z_like(p_sw_nw)
p_sw_se = p_sw - p_se; s_p_sw_se = z_like(p_sw_se)
p_se_ne = p_se - p_ne; s_p_se_ne = z_like(p_se_ne)
p_ne_nw = p_ne - p_nw; s_p_ne_nw = z_like(p_ne_nw)
p_se_nw = p_se - p_nw; s_p_se_nw = z_like(p_se_nw)
d_sw_nw = numpy.linalg.norm(p_sw_nw, axis=-1); s_d_sw_nw = z_like(d_sw_nw)
d_sw_se = numpy.linalg.norm(p_sw_se, axis=-1); s_d_sw_se = z_like(d_sw_se)
d_se_ne = numpy.linalg.norm(p_se_ne, axis=-1); s_d_se_ne = z_like(d_se_ne)
d_ne_nw = numpy.linalg.norm(p_ne_nw, axis=-1); s_d_ne_nw = z_like(d_ne_nw)
d_se_nw = numpy.linalg.norm(p_se_nw, axis=-1); s_d_se_nw = z_like(d_se_nw)
area_sw_se_nw = heron_area(d_sw_se, d_se_nw, d_sw_nw)
s_area_sw_se_nw = z_like(area_sw_se_nw)
```

```
area_se_ne_nw = heron_area(d_se_ne, d_ne_nw, d_se_nw)
s_area_se_ne_nw = z_like(area_se_ne_nw)
# Forward code ended with:
# return xyzs_wrapped, area_sw_se_nw.sum() + area_se_ne_nw.sum()
# We are merely interested in the sensitivity of the 2nd tuple entry
# (the area) on input parameters.
area = area_sw_se_nw.sum() + area_se_ne_nw.sum()
s_area_se_ne_nw += 1
s_area_sw_se_nw += 1
s1_d_se_ne, s1_d_ne_nw, s1_d_se_nw = grad_heron_area(
    d_se_ne, d_ne_nw, d_se_nw, s_area_se_ne_nw)
# Here, we are adding the gradients we received from grad_heron_area
# to our sensitivity-accumulators.
# There is an obvious improvement opportunity here:
# Re-write `grad_heron_area` in such a way that it receives our sensitivity-
# accumulator arrays and directly increments the sensitivities in these
# arrays, avoiding the copying.
s_d_se_ne += s1_d_se_ne
s_d_ne_nw += s1_d_ne_nw
s_d_se_nw += s1_d_se_nw
s1_d_sw_se, s1_d_se_nw, s1_d_sw_nw = grad_heron_area(
    d_sw_se, d_se_nw, d_sw_nw, s_area_sw_se_nw)
s_d_sw_se += s1_d_sw_se
s_d_se_nw += s1_d_se_nw
s_d_sw_nw += s1_d_sw_nw
# Backpropagating through numpy.linalg.norm():
# d{norm(vector)} / d{vector} = {vector}/{norm(vector)}
s_p_sw_nw += p_sw_nw * (s_d_sw_nw / d_sw_nw)[:, :, numpy.newaxis]
s_p_sw_se += p_sw_se * (s_d_sw_se / d_sw_se)[:, :, numpy.newaxis]
s_p_se_ne += p_se_ne * (s_d_se_ne / d_se_ne)[:, :, numpy.newaxis]
s_p_ne_nw += p_ne_nw * (s_d_ne_nw / d_ne_nw)[:, :, numpy.newaxis]
s_p_se_nw += p_se_nw * (s_d_se_nw / d_se_nw)[:, :, numpy.newaxis]
s_p_sw += s_p_sw_nw
s_p_nw -= s_p_sw_nw
s_p_sw += s_p_sw_se
s_p_se -= s_p_sw_se
s_p_se += s_p_se_ne
s_p_ne -= s_p_se_ne
s_p_ne += s_p_ne_nw
s_p_nw -= s_p_ne_nw
s_p_se += s_p_se_nw
s_p_nw -= s_p_se_nw
s_xyzs_wrapped[:-1, :-1, :] += s_p_sw
s_xyzs_wrapped[1:, :-1, :] += s_p_nw
s_xyzs_wrapped[:-1, 1:, :] += s_p_se
s_xyzs_wrapped[1:, 1:, :] += s_p_ne
if 'DDD':
```

```
  for delta, coords in ddd_delta_xyzs_wrapped:
    print(f'DDD s_xyzs_wrapped[{coords}] = {s_xyzs_wrapped[coords]}')
# Forward code:
# xyzs_wrapped = numpy.concatenate([xyzs, xyzs[:, :1, :]], axis=1)
s_xyzs += s_xyzs_wrapped[:, :-1, :]
s_xyzs[:, :1, :] += s_xyzs_wrapped[:, -1:, :]
# Forward code:
# xyzs = (  # shape [num_rings, num_vertices_per_ring, 3]
#     numpy.stack([xys_scaled_complex.real,
#                  xys_scaled_complex.imag,
#                  zs],
#                 axis=-1))
# Backpropagating gets interesting here, since we get to see 'sensitivity of
# the end result on the real/imaginary part of a complex quantity'.
# It so turns out that we have to take the complex conjugate here(!).
# This is discussed at the end of this notebook.
s_xys_scaled_complex += s_xyzs[..., 0] - 1j * s_xyzs[..., 1]
s_zs += s_xyzs[..., 2]
# Forward code:
# xys_scaled_complex = xys_unscaled_complex[numpy.newaxis, :] * (
#     rs[:, numpy.newaxis] + vertex_radial_deviations)
# The `xys_unscaled_complex` are just "constants" we do not need to backprop
# into.
# We perhaps should have introduced the 2nd factor as another intermediate
# quantity above, with its own sensitivity-accumulator, but since this
# subexpression is a simple sum that only shows up once, its
# sensitivity-accumulator only receives a single contribution, so we might
# just as well define it down here.
s_rs_plus_radial_deviations = s_xys_scaled_complex * (
    xys_unscaled_complex[numpy.newaxis, :])
# This is interesting: We just switched over to taking a product of
# complex sensitivities with a complex number - so we are now in
# complex-backprop land. Hence, the sensitivity introduced above is
# complex-valued, but we only want/need the real part here.
s_rs += s_rs_plus_radial_deviations.real.sum(axis=-1)
s_vertex_radial_deviations += s_rs_plus_radial_deviations.real
# This is the most important part of the "full gradient", which we are
# actually using in our computation.
# We could proceed to compute the r1/r2/z1/z2-sensitivities for the sake
# of completeness here. Physically, these would tell us about the force with
# which a soap film of constant surface tension would try to bring the
# rings together, respectively squeeze them.
# (This might be a Participant Exercise for very motivated participants,
# since backprop through numpy.linspace() will require a bit of thinking here,
# and potentially also debugging - so this could get somewhat frustrating
# when tried as an early exercise. If, however, we do not do this, then
# there also is no need to compute s_rs above.)
```

```
  s_inner_vertex_radial_deviations = s_vertex_radial_deviations[1:-1, :]
  return s_inner_vertex_radial_deviations
```

10.1.1 Hints for implementing grad_heron_area

(Not shown in the transcript, only in the accompanying notebooks.)

10.1.2 Testing it out

Let us actually try to gain some confidence in our gradient before we use it.

Our strategy: Define a random-but-reproducibly-so point and direction, define a 1-parameter function that walks along that direction, compare backprop-gradient with Taylor expansion.

[ ]:
```
def verify_grad_mesh_area(rng_seed=0):
  # Let us get a seeded guaranteed-reproducible-even-across-numpy-versions
  # random number generator.
  rng = numpy.random.RandomState(seed=rng_seed)
  # We are taking somewhat generic values for the problem.
  z1, z2, r1, r2 = -0.5, 2.0, 20.0, 21.0
  num_inner_rings, num_vertices_per_ring = 5, 10
  x0 = rng.normal(size=(num_inner_rings, num_vertices_per_ring), scale=0.1)
  x_dir_unnormalized = rng.normal(size=(num_inner_rings, num_vertices_per_ring))
  x_dir = x_dir_unnormalized / numpy.linalg.norm(x_dir_unnormalized)
  def f_area(s):
    _, area = vertices_and_total_mesh_area(z1, z2, r1, r2, x0 + s * x_dir)
    return area
  fprime_area_numerical = numerical_derivative(f_area)
  grad_at_x0 = grad_total_mesh_area(z1, z2, r1, r2, x0)
  return fprime_area_numerical(0), grad_at_x0.ravel().dot(x_dir.ravel())

# We compare 10 randomly picked positions and directions.
# If these gradients match, this provides rather strong validation.
for seed in range(10):
  print('seed:', seed, verify_grad_mesh_area(rng_seed=seed))

# Let us also quickly ad-hoc check our "Heron's area formula" gradient
# (To better illustrate this).
print('Heron-Check: area', heron_area(3.,4.,4.5),
      'numerical_grad[2]', (heron_area(3.,4.,4.5 + 1e-7) -
                            heron_area(3.,4.,4.5)) / 1e-7 ,
      'backprop_grad_full', grad_heron_area(3.,4.,4.5,1))
```

[ ]:
```
# The same demo as before - but this time, we report every call,
# rather than every 1000th.

demo2_opt_val, demo2_vertices = get_minimal_area(
    0.0, 18, 14.0, 14.0, 10, 20,
```

```
    report_every_n_calls=1,
    grad_area=grad_total_mesh_area)


pyplot2_axes = pyplot.axes(projection='3d')
pyplot2_axes.plot_wireframe(demo2_vertices[..., 0],
                            demo2_vertices[..., 1],
                            demo2_vertices[..., 2],
                            color='black')

# Participant Exercise:
# Compare total optimization run time for the previous and this exercise.

# Participant Exercise:
# How much finer can we make the discretization?
# At how many degrees of freedom does this get "high effort"?

# Participant Exercise:
# Put a numerical bug(!) into the gradient-of-total-area function in some
# random place (such as: change a sign from + to -)
# and check how this then impacts optimization here.

# Participant Exercise:
# Adjust the code to allow other boundary shapes than the edges of a symmetric
# conical frustum. (Such as: using different semimajor axes,
# adding an in-plane displacement, etc.)
```

11 A first Machine Learning Example

Let us actually use what we learned to build and train an extremely simple machine learning model without using any ML or backpropagation framework library (apart from using TensorFlow to load the example data).

For quite a few years, the "fruit fly" / "hydrogen problem" / "N=4 SYM" type system used for ML was handwritten digit classification using the "MNIST" dataset of digitized handwriting samples - tens of thousands of digits written by real people and digitized to 28x28 pixel grayscale resolution, with center-of-gravity for the ink shifted-to-center and some other such normalizations.

One of the big problems with MNIST is that it is "too easy to learn". Nowadays, it is only used rarely, and mostly in the context of unusual architectures or applications. Nevertheless, knowing about this dataset is important background for reading some of the older foundational papers.

Let us build a very primitive classifier that tells us "Yes" or "No", depending on whether or not it thinks the digit is an 8. We will do this not on 28x28 images, but on images downsampled to 14x14. Also, we will only use 1000 example images for training - which everybody would recognize in these days as "being way too few examples to meaningfully do ML".

But first, let us load and inspect this popular dataset.

[ ]:
```
import tensorflow_datasets as tfds

(ds_train, ds_test), ds_info = tfds.load(
    'mnist',
    split=['train', 'test'],
    shuffle_files=False,
    as_supervised=True,
    with_info=True,
)

print(ds_info)
```

```
tfds.core.DatasetInfo(
    name='mnist',
    full_name='mnist/3.0.1',
    description="""
    The MNIST database of handwritten digits.
    """,
    homepage='http://yann.lecun.com/exdb/mnist/',
    data_path='/root/tensorflow_datasets/mnist/3.0.1',
    file_format=tfrecord,
    download_size=11.06 MiB,
    dataset_size=21.00 MiB,
    features=FeaturesDict({
        'image': Image(shape=(28, 28, 1), dtype=uint8),
        'label': ClassLabel(shape=(), dtype=int64, num_classes=10),
    }),
    supervised_keys=('image', 'label'),
    disable_shuffling=False,
    splits={
        'test': <SplitInfo num_examples=10000, num_shards=1>,
        'train': <SplitInfo num_examples=60000, num_shards=1>,
    },
    citation="""@article{lecun2010mnist,
      title={MNIST handwritten digit database},
      author={LeCun, Yann and Cortes, Corinna and Burges, CJ},
      journal={ATT Labs [Online]. Available: http://yann.lecun.com/exdb/mnist},
      volume={2},
      year={2010}
    }""",
)
```

[ ]:
```
# We actually take 4000 image samples from the "MNIST training set"
# and split them into two sets, the "even indices" and "odd indices",
# where we use one of these as our reduced-size training set,
# and the other one to assess how well our model manages to handle
# input it has never seen before.
images_and_labels_all = list(ds_train.take(4000).as_numpy_iterator())
```

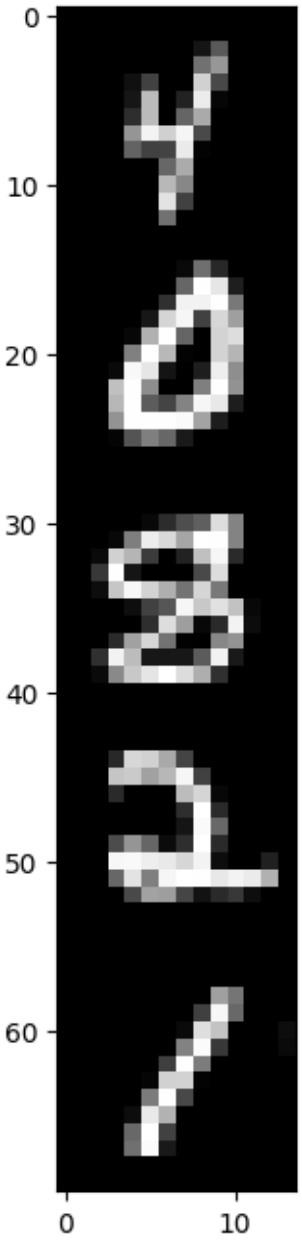
```
images_and_labels_training = images_and_labels_all[::2]
images_and_labels_test = images_and_labels_all[1::2]

def as_14x14(data):
  """Maps a [..., 28, 28]-array to a [..., 14, 14]-array by 2x2-averaging."""
  data = numpy.asarray(data)
  shape = data.shape
  return data.reshape(shape[:-3] + (14, 2, 14, 2)).mean(axis=(-1, -3))

# Raw data use unsigned 8-bit grayscale values (integers in [0..255]).
# Rescale to floats in 0..1 and stack them up into a [num_examples, 28, 28]
# array.
images_training = numpy.stack(
    [image / 255.0 for image, label in images_and_labels_training],
    axis=0)
labels_training = numpy.array(
    [label for image, label in images_and_labels_training])
images_test = numpy.stack(
    [image / 255.0 for image, label in images_and_labels_test],
    axis=0)
labels_test = numpy.array(
    [label for image, label in images_and_labels_test])

pyplot.imshow(as_14x14(images_training[:5]).reshape(-1, 14) , cmap='gray')
pyplot.show()
print(labels_training[:5])

# Let us see how many 8-digits there are in this sample:
print('Number of 8-digits:', sum(x==8 for x in labels_training))
```




```
[4 0 8 2 1]
Number of 8-digits: 209
```

Overall, we have 1000 training images which we will downscale to 14x14=196 brightness float-values (one per pixel). So, our reduced-size dataset consists of 39200 floating point numbers in total.

There are many different approaches one could try to come up with a digit-8 classifier, such as for example:

1. Write tailored code that exploits special properties of the digit "8", such as:

"Classify this digit as 8 if and only if there are three separate non-inked regions".

2. Consider each example as a point in high-dimensional (D=$14^2 = 196$) space and, for a given input, look at the classifications of the 10 nearest-neighbor examples.

We will look into these later. Approach (1.) was used somewhat widely in the 90s for problems "where the best we can do is ad hoc heuristics". ML follows the ideas presented in Turing's 1948 paper "Intelligent Machinery [16]": "can we use examples to fine-tune parameters of a general architecture via optimization, in order to maximize classifier performance (w.r.t. the capabilities of the given architecture)?"

```
A great positive reason for believing in the possibility of making thinking
machinery is the fact that it is possible to make machinery to imitate any
small part of a man. That the microphone does this for the ear, and the
television camera for the eye are commonplaces. (...) Here we are chiefly
interested in the nervous system. We could produce fairly accurate electrical
models to copy the behaviour of nerves, but there seems very little point in
doing so. It would be rather like (...) cars which walked on legs (...).

(...)

If we are trying to produce an intelligent machine, and are following the
human model as closely as we can, we should begin with a machine with very
little capacity to carry out elaborate operations or to react in a disciplined
manner to orders (taking the form of inference). Then by applying appropriate
inference, mimicking education, we should hope to modify the machine until it
could be relied on to produce definite reactions to certain commands.
```

What is the "dumbest" possible idea we can come up with here that still looks somewhat reasonable? We want a function that maps input data, as a $\mathbb{R}^{14\times 14}$ vector, to some sort of "accumulated evidence that we are looking at a number 8 digit".

The structurally simplest functions we can think of doing something like this are linear (rather, affine, i.e. "with offset" functions. So, we are looking for a linear function that we want to output some large positive number if we are looking at a digit 8 and some large negative number if not. But what would our objective function for numerical minimization then look like?

Right now, our focus is on "numerical optimization" and "doing everything from scratch, no ML frameworks allowed", so we will have to come back to this question later in the course. Here, we will only use two ideas - to be justified later.

1. For every given input image, we want to map the accumulated evidence to a "this is how I, the classifier, would bet" (Bayesian) probability for the example to have a positive (binary classification) label.

   It so turns out that the appropriate function to do this is the "sigmoid function" $f(x) = 1/(1+\exp(k-x))$, which we of course also are familiar with in the context of the thermodynamics of a 2-state system. Here, the offset is a tunable/"learnable" parameter.

2. Given probabilities and ground truth labels, we need to produce a "how far are we off on these examples" score. Here, we go with a smooth variant of the basic idea that "for misclassifications, we want to count the number of interval bisections that tell us how far off our

probability estimate was". So, if we have a label=True case where we say "p=6.25% this is True", this needs "3 bisecting binary questions" (50% -> 25% -> 12.5% -> 6.25%), while if we say "p=25%", this needs "1 bisecting binary question" so is only 1/3 as bad overall in terms of information content.

Let's build this.

[ ]:
```
def get_digit8_classifier(training_images,
                          training_labels,
                          seed=0,
                          maxiter=10_000,
                          debug_print_every_n_calls=100):
  """Returns a pair (classifier_parameters, classifier) for detecting 8-s.

  Args:
    training_images:
      [0..1]-float [num_images, num_rows, num_columns]-array-like,
      the training dataset.
    training_labels: [num_images]-array-like, array of ground truth labels
      (0-9) for the images in `training_images`.
    seed: Random number generator seed.
    maxiter: Maximal number of iterations for the BFGS-minimizer.
    debug_print_every_n_calls: Controls how often we print progress information.

  Returns:
    A pair of `(classifier_parameters_vector, classifier_callable)`,
    where calling `classifier_callable` on a
    [num_images_to_classify, num_rows, num_columns]-array-like collection
    of image data returns a corresponding vector of best-estimate probabilities
    for the image to show an 8-digit.
  """
  training_images = numpy.asarray(training_images)
  training_labels = numpy.asarray(training_labels)
  label_positive = training_labels == 8
  def get_prob(params, images):
    stencil_params = params[:-1]
    offset = params[-1:]  # A [1]-vector, for broadcasting.
    evidence = numpy.einsum('ix,x->i',
                            images.reshape(images.shape[0], -1),
                            stencil_params)
    return 1 / (1 + numpy.exp(offset-evidence))
  num_calls = 0
  def get_quality_score(params):
    nonlocal num_calls
    prob = get_prob(params, training_images)
    # The 1e-12 offsets help handling cases where exp(-evidence) is numerically
    # zero due to underflow. This avoids running into log(0), which produces
    # NaN intermediate results.
```

```
  result = -(label_positive * numpy.log(1e-12 + prob) +
             (1 - label_positive) * numpy.log(1+1e-12-prob)).mean()
  num_calls += 1
  if (debug_print_every_n_calls is not None and
      num_calls % debug_print_every_n_calls == 0):
    print(f'N={num_calls:6d}: score={result:16.10f}')
  return result
# BEGIN Gradient code
# We want the sensitivity of `prob` on parameters.
def get_s_prob(params, images, s_result):
  stencil_params = params[:-1]
  offset = params[-1:]  # A [1]-vector, for broadcasting.
  s_params = numpy.zeros_like(params)
  evidence = numpy.einsum('ix,x->i',
                          images.reshape(images.shape[0], -1),
                          stencil_params)
  exp_neg_evidence = numpy.exp(offset-evidence)
  denom = 1 + exp_neg_evidence
  result = 1 / denom
  #
  # s_denom = -s_result / (denom * denom)
  s_exp_neg_evidence = -s_result / (denom * denom)
  s_evidence = s_exp_neg_evidence * (-exp_neg_evidence)
  s_offset = -s_evidence.sum(keepdims=True)
  s_stencil_params = numpy.einsum('ix,i->x',
                                  images.reshape(images.shape[0], -1),
                                  s_evidence)
  return numpy.concatenate([s_stencil_params, s_offset])
def get_s_quality_score(params):
  prob = get_prob(params, training_images)
  s_prob = numpy.zeros_like(prob)
  part1 = label_positive * numpy.log(1e-12 + prob)
  part2 = (1-label_positive) * numpy.log(1+1e-12-prob)
  result = -(part1 + part2).mean()
  #
  s_part1 = -1 / len(part1)
  s_part2 = -1 / len(part2)
  s_prob += s_part1 * label_positive / (1e-12 + prob)
  s_prob -= s_part2 * (1-label_positive) / (1+1e-12-prob)
  return get_s_prob(params, training_images, s_prob)
# END Gradient code
rng = numpy.random.RandomState(seed=seed)
# Model parameters are: 14x14 per-pixel-evidence weights, plus one overall
# evidence-offset.
v_start = rng.normal(size=14*14 + 1, scale=1e-2)
opt = scipy.optimize.fmin_bfgs(get_quality_score,
                               v_start,
```

```
                                  fprime=get_s_quality_score,
                                  gtol=1e-10,
                                  maxiter=maxiter)
  return opt, lambda images: get_prob(opt, images)

# Quite likely, training/optimization will stop here once we run into
# exp() overflow - but even with optimization not proceeding to the minimum,
# we do get a somewhat useful classifier.
opt_params, classifier = get_digit8_classifier(as_14x14(images_training),
                                                labels_training)
```

```
N=   100: score=    0.0747413554
N=   200: score=    0.0651305360
N=   300: score=    0.0620802156
N=   400: score=    0.0607213317
N=   500: score=    0.0600792142
N=   600: score=    0.0596360551
N=   700: score=    0.0591480792
N=   800: score=    0.0588810554
N=   900: score=    0.0585761021
N=  1000: score=    0.0580431305
N=  1100: score=    0.0578176706
N=  1200: score=    0.0576444457
N=  1300: score=    0.0574787213
N=  1400: score=    0.0573991570
N=  1500: score=    0.0573255619
N=  1600: score=    0.0572122781
N=  1700: score=    0.0569957265
```

```
<ipython-input-68-cbe0ab0e1b3e>:63: RuntimeWarning: overflow encountered in
multiply
  s_exp_neg_evidence = -s_result / (denom * denom)
```

```
Warning: Desired error not necessarily achieved due to precision loss.
         Current function value: 0.056684
         Iterations: 1751
         Function evaluations: 1768
         Gradient evaluations: 1756
```

```
<ipython-input-68-cbe0ab0e1b3e>:34: RuntimeWarning: overflow encountered in exp
  return 1 / (1 + numpy.exp(offset-evidence))
<ipython-input-68-cbe0ab0e1b3e>:58: RuntimeWarning: overflow encountered in exp
  exp_neg_evidence = numpy.exp(offset-evidence)
<ipython-input-68-cbe0ab0e1b3e>:64: RuntimeWarning: invalid value encountered in
multiply
  s_evidence = s_exp_neg_evidence * (-exp_neg_evidence)
<ipython-input-68-cbe0ab0e1b3e>:34: RuntimeWarning: overflow encountered in exp
  return 1 / (1 + numpy.exp(offset-evidence))
<ipython-input-68-cbe0ab0e1b3e>:58: RuntimeWarning: overflow encountered in exp
```

```
  exp_neg_evidence = numpy.exp(offset-evidence)
<ipython-input-68-cbe0ab0e1b3e>:63: RuntimeWarning: overflow encountered in
multiply
  s_exp_neg_evidence = -s_result / (denom * denom)
<ipython-input-68-cbe0ab0e1b3e>:64: RuntimeWarning: invalid value encountered in
multiply
  s_evidence = s_exp_neg_evidence * (-exp_neg_evidence)
```

We do observe that optimization failed here since we were not appropriately careful implementing our objective function in a way that does not readily run into numerical pathologies. It is perhaps useful to show that such things can and do happen, and thinking about how to avoid numerical cancellation, blow-ups etc. is generally important when implementing machine learning components.

We could fix this here, but do not really have to - optimization, even as it technically failed to complete, has already produced a somewhat useful performance-enhancement. Had we used a loss function from a ML library, this point would have been addressed in its implementation.

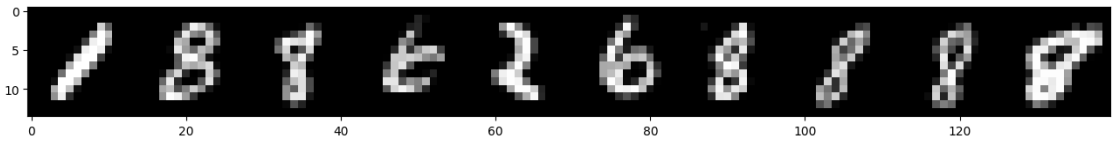
[ ]:
```
# Let us actually try out our classifier on some of our test images:
num_test = 2000
max_num_bad_to_show = 20
num_bad_shown = 0
images_test_14x14 = as_14x14(images_test[:num_test])
classified_test_images = classifier(images_test_14x14)
bad_labels_images = []
for n, prob, label_test in zip(range(num_test),
                               classified_test_images,
                               labels_test):
  is_bad = (prob > 0.5) != int(label_test == 8)
  # Disable this check to print all classifications.
  if is_bad and num_bad_shown < max_num_bad_to_show:
    num_bad_shown += 1
    print(f'N={n:3d}: label={label_test}, '
          f'prob={prob:.10f} {"X" if is_bad else " "}')
  if is_bad:
    bad_labels_images.append((label_test, images_test_14x14[n]))
pyplot.imshow(
    numpy.concatenate(
        [image for label, image in bad_labels_images[:10]],
        axis=1),
    cmap='gray')
pyplot.show()
bad_labels = [label for label, image in bad_labels_images]

print(len(bad_labels), bad_labels)

# Let us also check if we are actually better than always-guessing-"not an 8".
num_test_8s = (labels_test[:num_test] == 8).sum()
sigma = (num_test * 0.1 * 0.9)**.5
```

```
print('\n###\n'
      f'Always-guessing-not-8 would make {num_test_8s} misclassifications '
      f'({100 * num_test_8s / num_test:.1f}%).\n'
      f'We made {len(bad_labels)} misclassifications '
      f'({100 * len(bad_labels) / num_test:.1f}%).\n'
      f'With sigma = sqrt({num_test} * 0.1 * 0.9) = {sigma:.2f}, this is at\n'
      f'{abs(len(bad_labels) - num_test_8s) / sigma:.2f} sigma for the '
      f'baseline classifier.')
```

```
N=  9: label=1, prob=0.9785949016 X
N= 31: label=8, prob=0.0017661758 X
N= 97: label=8, prob=0.4151601095 X
N=129: label=6, prob=0.6621850765 X
N=163: label=2, prob=0.7174619928 X
N=169: label=6, prob=0.5418635124 X
N=170: label=8, prob=0.0022988911 X
N=184: label=8, prob=0.3697129140 X
N=187: label=8, prob=0.2133721492 X
N=212: label=8, prob=0.0934954688 X
N=227: label=8, prob=0.0000000000 X
N=228: label=8, prob=0.0000000000 X
N=262: label=8, prob=0.1821384720 X
N=282: label=3, prob=0.9977189467 X
N=289: label=5, prob=0.9264995652 X
N=290: label=8, prob=0.0000002562 X
N=291: label=8, prob=0.0000000032 X
N=293: label=8, prob=0.0000000000 X
N=307: label=8, prob=0.0574443067 X
N=318: label=9, prob=0.7543842702 X
```




```
121 [1, 8, 8, 6, 2, 6, 8, 8, 8, 8, 8, 8, 8, 3, 5, 8, 8, 8, 8, 9, 0, 4, 8, 0, 8,
8, 0, 2, 9, 1, 0, 1, 2, 8, 8, 5, 8, 0, 8, 8, 8, 2, 5, 2, 8, 4, 2, 8, 4, 5, 8, 6,
7, 4, 4, 2, 9, 8, 8, 3, 8, 0, 8, 8, 8, 8, 7, 5, 5, 8, 9, 0, 1, 8, 5, 8, 8, 8, 2,
4, 8, 8, 8, 7, 2, 8, 4, 5, 5, 8, 1, 8, 8, 8, 8, 5, 2, 8, 8, 8, 2, 8, 8, 8, 2,
8, 8, 8, 8, 5, 8, 1, 8, 8, 4, 5, 8, 5, 4, 2]

###
Always-guessing-not-8 would make 203 misclassifications (10.2%).
We made 121 misclassifications (6.0%).
With sigma = sqrt(2000 * 0.1 * 0.9) = 13.42, this is at
6.11 sigma for the baseline classifier.
```

11.1 Participant Exercises

- On the MNIST dataset, humans are observed to have a "which digit is it" classification error rate in the ballpark of 2-3%. Change the code to load 500 examples and visualize them in a 50x10 grid. Hand-classify them and compare your classifications with the "ground truth" labels. Is this claimed human error rate plausible?
- Above, we introduced an 'offset'. Does this parameter actually have an impact on performance? Replace `offset` with `0 * offset` in the code above to check this.
- Find out if the MNIST training dataset (50_000 examples) contains any exact duplicate images. (This might need some creativity.)
- Change the above code to only see examples where the ground truth label is "1" or "2", and build a binary "is it a digit 1?" classifier. Is it much better than random guessing? [It should be!]
- Can you fix the "numerical overflow" problem observed above?

One way to think about such classification problems is as follows: We decided to believe in the existence of two smooth $\mathbb{R}^{14\times14} \mapsto \mathbb{R}$ probability distribution functions which describe the probability-density for "a human asked to write a digit 1 (respectively, 2) will produce something that looks like this". We do not know what they look like, but we can sample thousands of the examples from these probability distributions - which we did.

When we classify a never-seen-before example, we try to use what we have seen to estimate the likelihood ratio "is this a 1" vs. "is this a 2", and make a call based on this estimate.

Let us try a radically different approach that does not really use any "learning": For a given not-observed-before input, we determine the euclidean distance to each training set example, and look at the N closest neighbors and their classes.

[ ]:
```
def get_neighborhood_analyzer(images_training, labels_training):
  images_training = (
      numpy.asarray(images_training).reshape(images_training.shape[0], -1))
  labels_training = numpy.asarray(labels_training)
  def get_neighborhood(image_new, num_neighbors=10):
    image_new = numpy.asarray(image_new).reshape(1, -1)
    distances = numpy.linalg.norm(images_training - image_new, axis=-1)
    sorted_indexed_distances = sorted(enumerate(distances), key=lambda nd: nd[1])
    return [(dist, labels_training[index])
            for index, dist in sorted_indexed_distances[:num_neighbors]]
  return get_neighborhood

analyzer = get_neighborhood_analyzer(as_14x14(images_training), labels_training)
for n in range(20):
  analyzed = [(numpy.round(dist, 2), label)
              for dist, label in analyzer(as_14x14(images_test[n]))]
  print(n, labels_test[n], analyzed)

# Participant Exercise:
```

```
# Build a 1-out-of-10 classifier out of such a neighborhood-analyzer by
# doing some (distance-weighted?) counting of the labels of the 10 (or 100)
# nearest neighbors in the training dataset.
```

```
0 1 [(0.6, 1), (0.94, 1), (0.96, 1), (1.0, 1), (1.15, 1), (1.17, 1), (1.21, 1),
(1.21, 1), (1.22, 1), (1.28, 1)]
1 7 [(2.24, 7), (2.25, 7), (2.35, 7), (2.37, 7), (2.39, 7), (2.39, 7), (2.44,
7), (2.47, 7), (2.47, 9), (2.49, 7)]
2 1 [(1.06, 1), (1.13, 1), (1.2, 1), (1.24, 1), (1.24, 1), (1.24, 1), (1.3, 1),
(1.32, 1), (1.34, 1), (1.36, 1)]
3 7 [(1.57, 7), (1.8, 7), (1.8, 7), (1.86, 7), (2.0, 7), (2.16, 7), (2.26, 7),
(2.36, 7), (2.36, 7), (2.38, 7)]
4 6 [(1.55, 6), (1.69, 6), (1.72, 6), (1.92, 6), (1.96, 6), (1.98, 6), (2.01,
6), (2.01, 6), (2.16, 6), (2.17, 6)]
5 4 [(2.19, 4), (2.28, 4), (2.67, 4), (2.84, 4), (2.89, 5), (2.95, 4), (2.99,
9), (3.05, 4), (3.07, 4), (3.08, 4)]
6 7 [(1.89, 9), (1.92, 7), (1.92, 7), (1.94, 7), (2.14, 1), (2.16, 7), (2.22,
9), (2.23, 7), (2.24, 7), (2.26, 9)]
7 3 [(1.86, 3), (2.27, 3), (2.35, 3), (2.45, 3), (2.5, 3), (2.56, 3), (2.58, 3),
(2.6, 3), (2.62, 3), (2.7, 3)]
8 9 [(2.28, 9), (2.4, 9), (2.45, 9), (2.56, 9), (2.56, 3), (2.6, 9), (2.63, 9),
(2.67, 9), (2.68, 9), (2.68, 9)]
9 1 [(1.23, 1), (1.4, 1), (1.4, 1), (1.49, 1), (1.62, 1), (1.72, 1), (1.73, 1),
(1.91, 1), (2.04, 1), (2.08, 1)]
10 6 [(1.6, 6), (1.64, 6), (1.88, 6), (1.91, 6), (1.94, 6), (1.97, 6), (1.97,
6), (1.99, 6), (2.0, 6), (2.04, 6)]
11 9 [(1.6, 9), (2.17, 9), (2.29, 9), (2.9, 4), (2.95, 4), (2.96, 9), (3.06, 9),
(3.11, 9), (3.12, 4), (3.14, 9)]
12 4 [(1.57, 4), (1.67, 4), (1.93, 4), (2.0, 9), (2.04, 4), (2.13, 4), (2.26,
9), (2.27, 9), (2.27, 9), (2.29, 9)]
13 9 [(1.56, 9), (1.63, 7), (1.68, 9), (1.7, 9), (1.73, 9), (1.73, 9), (1.87,
9), (1.87, 9), (1.91, 9), (1.96, 9)]
14 7 [(1.4, 7), (1.6, 7), (1.64, 7), (1.81, 7), (1.83, 7), (1.84, 7), (1.87, 7),
(1.91, 7), (1.91, 7), (1.97, 7)]
15 3 [(2.41, 8), (2.44, 3), (2.48, 3), (2.49, 3), (2.52, 3), (2.53, 3), (2.57,
3), (2.71, 3), (2.72, 3), (2.72, 8)]
16 9 [(2.76, 9), (2.8, 2), (2.85, 9), (2.88, 9), (2.97, 9), (3.06, 9), (3.1, 9),
(3.11, 7), (3.12, 9), (3.13, 7)]
17 9 [(2.52, 9), (2.59, 9), (2.7, 9), (2.7, 7), (2.71, 9), (2.73, 9), (2.73, 9),
(2.76, 9), (2.79, 4), (2.84, 9)]
18 6 [(2.29, 6), (2.45, 6), (2.67, 6), (2.72, 6), (2.78, 6), (2.79, 6), (2.82,
6), (2.82, 6), (2.82, 6), (2.83, 6)]
19 4 [(2.17, 4), (2.27, 4), (2.36, 4), (2.64, 4), (2.65, 4), (2.68, 4), (2.72,
4), (2.74, 9), (2.76, 4), (2.79, 4)]
```

Overall, what we have been doing here, especially with this last nearest-neighbors approach, has some smell of "having made many ad-hoc and arbitrary choices". For example, the distribution of brightness-values observed on the training set for a near-center pixel will in general be a rather

different one than for a pixel that only very rarely receives ink - and yet, "we treat them all the same w.r.t. measuring distance" - is this actually justified?

Clearly, there are numerous ways to tune and tweak any such ML-ish approach, and (unfortunately) a widely held view in the ML community is that a criterion for publication-worthiness of research is whether it meaningfully manages to improve performance numbers (such as: fraction of correct classifications) on some of the (more or less) widely discussed model problems/datasets. Due to this situation, it can be both very easy to get published in ML but at the same time accept/reject decisions on papers do strongly depend on chance (a large body of articles in need of review naturally impacts the quality of reviews).

It has been said that "ML exists because humans are not good at seeing structure in very high dimensions", and "ML works" because real-world probability distributions may have some subtle structure but in general are not as badly behaved as they could be in principle. In this sense, we can see ML like a "telescope like perception-enhancing tool, but for seeing (coarse) structure in very high dimensions".

So, much of ("supervised", i.e. "ground truth target labels based") Machine Learning is about finding "tight" approximations to high dimensional probability distributions. Depending on how we build our parametric model, we can express only some structural features but not others. For our optimizer-based classifier above, the "language" of likelihood-ratio functions expressible by our ansatz has a lot of extra constraining structure - if we undo the final sigmoid that gives us a probability, we find a function that only linearly depends on the brightness ("ink value") of each pixel. This means especially: If we start from a blank image, then there will be a way to add ink that increases the digit-8-evidence fastest, and along every direction that has a positive scalar product with this one, we will add figure-8-evidence. It might just be that we "hit maximum brightness" on some pixel before we build up any meaningful such evidence. In particular, there is no way to represent any "if these two pixels have ink, that correlation provides us extra evidence" type of information in this overly simple model. We could of course amend our model, introducing extra (tunable) parameters, that are sensitive to such correlations.

Ultimately, this class of ML applications revolve around: "If we get to see examples drawn from a distribution that is not too different from what we obtained the training set from, what can we achieve by trying to maximize the odds of getting our predictions right?" In a way, this essentially statistics-based approach is very different from what one would intuitively call "learning", and this is nicely illustrated by training (i.e. parameter-tuning) a ML model on the training set but then feeding it trivially-transformed input, such as handwritten digits that have been rotated by 90 degrees. Clearly, "learning" is about as much a different notion in ML and human education context as "force" has different meanings in physical and colloquial context.

This "most likely the thing we want according to the background distribution" approach can have quite interesting - and sometimes problematic - consequences, especially if examples codify some form of "established bad habits" which then get baked into models. On Google Shopping, it is no problem to buy "two inch nails", but basically impossible to buy "nine inch nails", since that also happens to be the name of a popular band.

The classifier we built is simple enough to do one more thing with it: Let us actually plot the per-pixel-digit-8-evidence-weights themselves, reshaped to 14x14. This tells us "the direction in $\mathbb{R}^{14\times 14}$ along which evidence-for-digit-8 increases fastest".

[ ]:
```
# Looking at numerical values:
pprint.pprint(opt_params[:-1].round(2).reshape(14, 14).tolist(), width=200)
# "Clipping outliers" which provide a lot of evidence.
pyplot.imshow(numpy.clip(opt_params[:-1].reshape(14, 14), -10, 10), cmap='cool')
pyplot.show()
#
# We observe that there are some pixels 'just outside where a figure-8
# would normally have ink' that yield strong negative evidence, while some
# pixels that sit where we naturally would expect a figure-8 to be
# to provide mild positive evidence, the more the less likely any of the other
# digits are to have ink there.

# Let us build an alternative training set that consists of digits-8 plus
# "noise with statistically the same pixel-brightness distribution".
training_14x14 = as_14x14(images_training)
# These are only the digits-8.
training_14x14_digit8 = training_14x14[labels_training == 8]
# We start by repeating this dataset of digits-8 nine times.
# On these copies, we randomly shuffle the 196 pixels, so we get "noise images"
# which however have the same ink-distributions as our digit-8 samples.
training_14x14_clones = numpy.tile(training_14x14_digit8, [9, 1, 1])
rng = numpy.random.RandomState(seed=0)  # For reproducibility.
# Per-pixel ink-values are in [0..1]. We rescale to [0..0.5], then add a
# random integer, rescale to [num_samples, 14*14], sort along the total-pixels
# axis according to ink-value, and throw away the integral part and scale back.

random_14x14 = (
     (0.5 * training_14x14_clones).reshape(-1, 14*14) +
     rng.randint(1, 10**9, size=(training_14x14_clones.shape[0], 14*14)))
random_14x14.sort(axis=-1)
training_random_14x14 = numpy.concatenate(
    [training_14x14_digit8,
     (random_14x14.reshape(-1, 14, 14) % 1) * 2],
     axis=0)

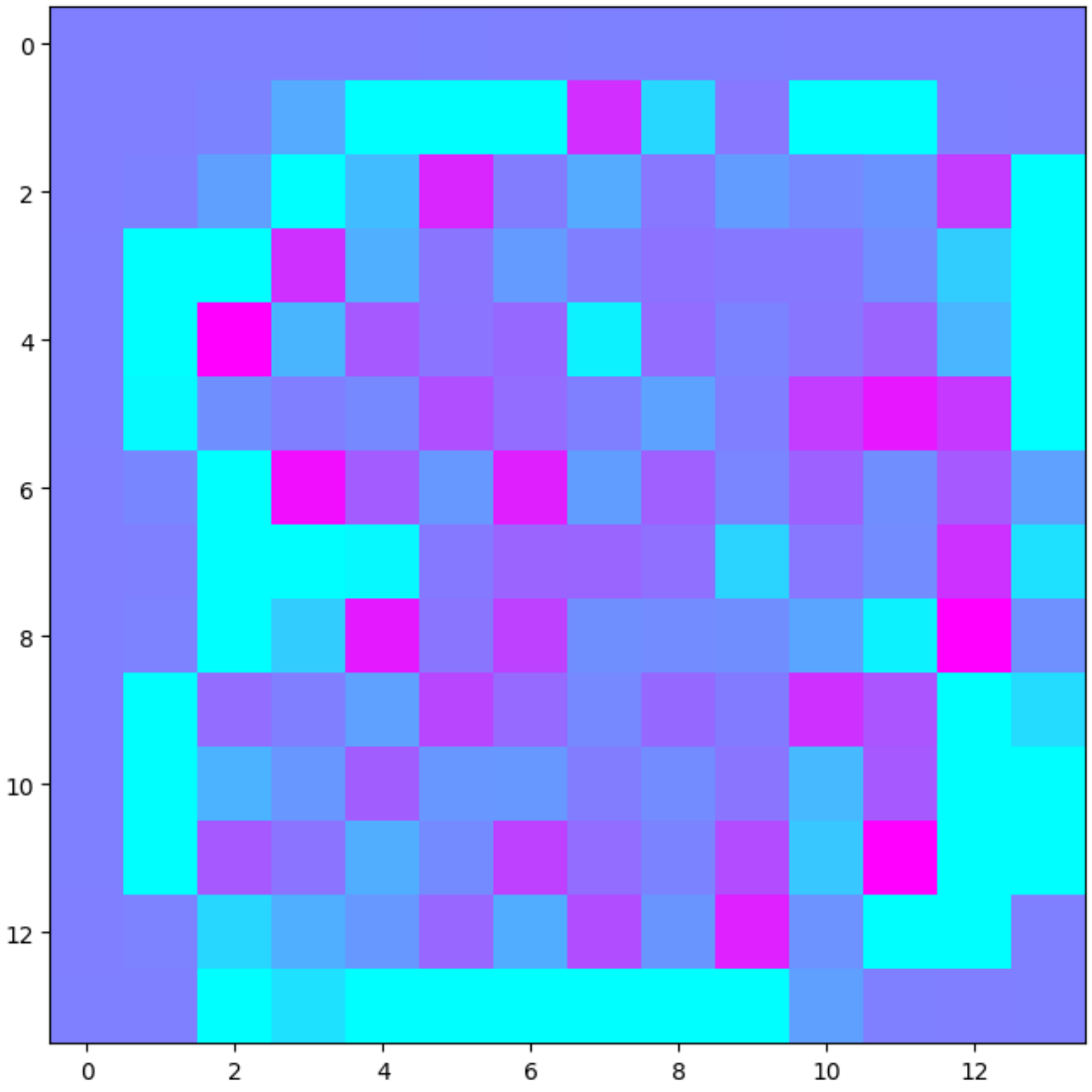
pyplot.imshow(training_random_14x14[-1], cmap='gray'); pyplot.show()

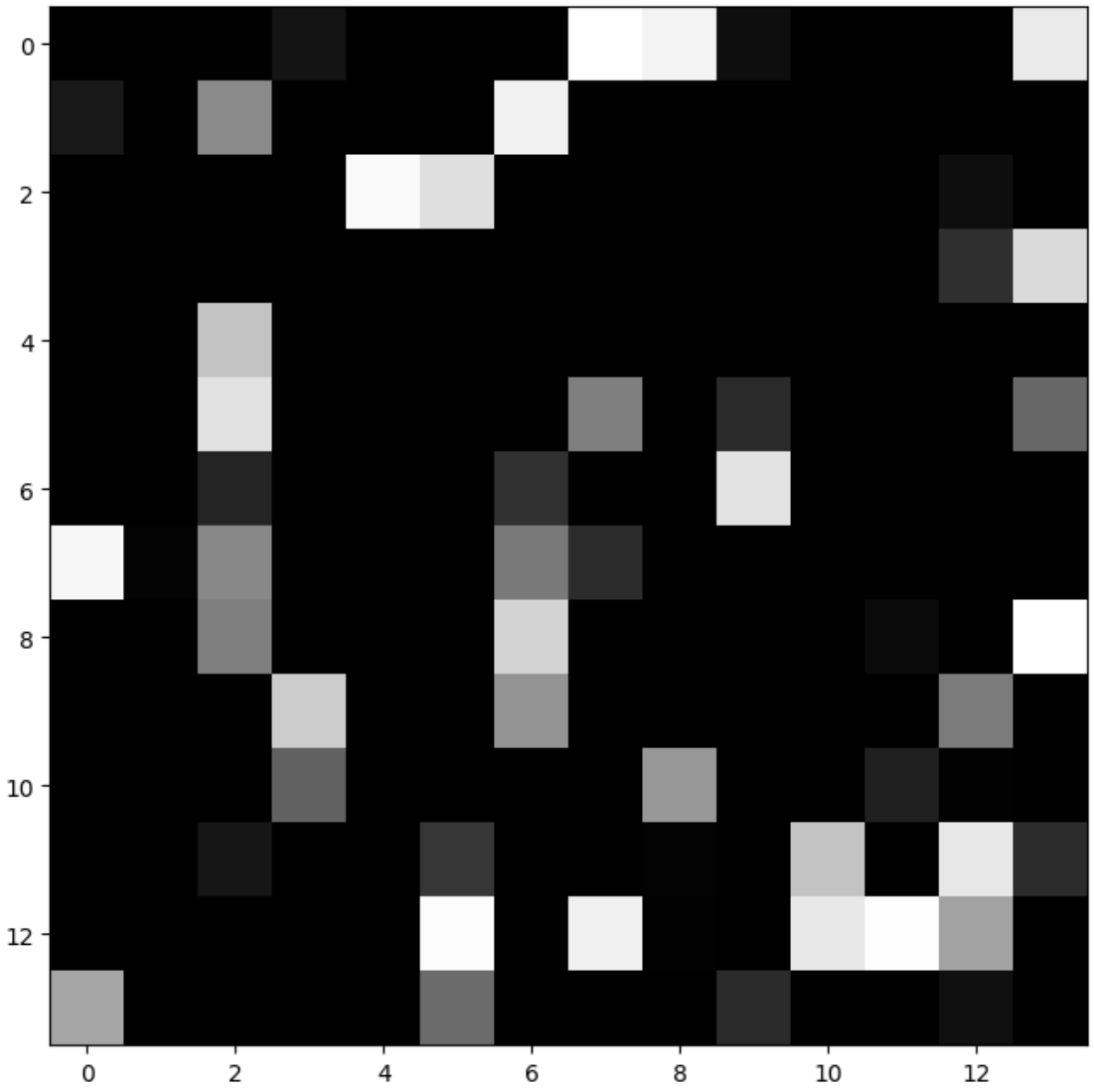

opt_params_rand, classifier_rand = get_digit8_classifier(
    training_random_14x14,
    [8] * len(training_14x14_digit8) + [0] * len(training_14x14_digit8) * 9,
    debug_print_every_n_calls=1)

# Showing the per-pixel-evidence-contribution for this classifier would
# be naturally expected to give us something like an "averaged digit-8".
# This is indeed what happens here.
pyplot.imshow(numpy.clip(opt_params_rand[:-1].reshape(14, 14), -10, 10),
```

```
              cmap='cool')
pyplot.show()
```

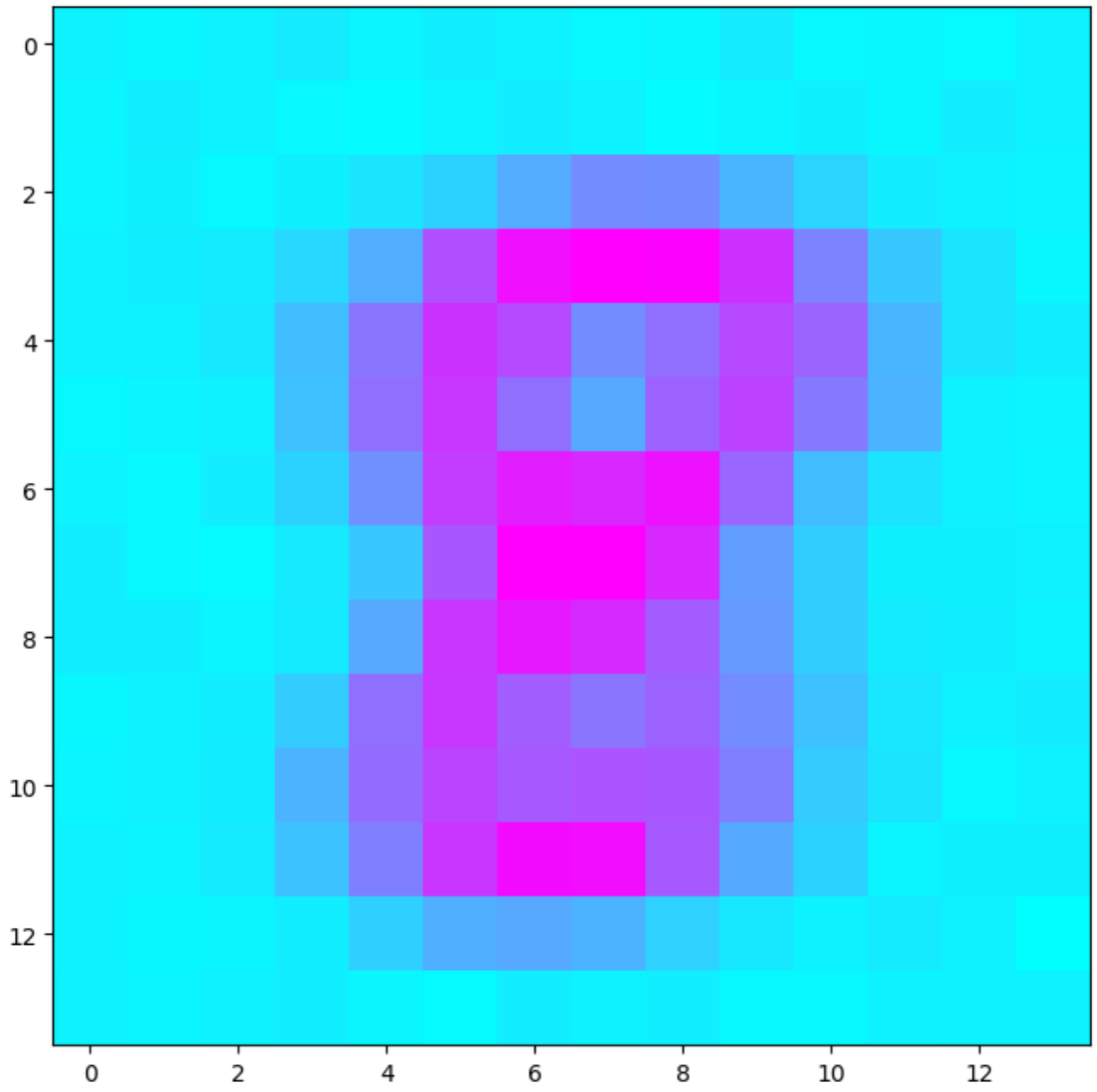
```
[[0.02, 0.0, 0.01, 0.02, 0.01, -0.09, -0.02, -0.12, -0.04, -0.0, 0.0, 0.01,
0.01, 0.0],
 [0.0, 0.0, -0.24, -3.42, -40.18, -120.73, -574.21, 6.38, -6.84, 0.56, -25.44,
-23.46, -0.09, -0.0],
 [0.02, -0.09, -2.5, -71.73, -4.82, 7.0, 0.16, -3.4, 0.65, -2.22, -0.84, -1.58,
5.17, -132.14],
 [-0.02, -22.14, -41.19, 6.13, -3.71, 0.8, -2.13, 0.06, 0.97, 0.51, 0.5, -1.09,
-6.02, -14.92],
 [-0.01, -30.91, 11.39, -4.16, 3.03, 0.89, 1.81, -9.05, 1.41, -0.3, 0.76, 2.18,
-4.29, -70.48],
 [-0.06, -9.57, -1.2, 0.01, -0.76, 3.76, 1.43, -0.01, -2.69, 0.06, 5.22, 8.11,
5.48, -26.5],
 [0.01, -0.55, -23.04, 8.86, 2.74, -1.91, 7.52, -2.33, 2.5, -0.53, 2.29, -1.14,
3.02, -2.64],
 [0.0, -0.04, -148.09, -15.03, -9.45, 0.45, 2.13, 2.04, 1.23, -6.61, 0.68,
-0.97, 6.07, -7.64],
 [-0.01, -0.2, -109.31, -5.99, 7.9, 0.85, 4.81, -1.19, -0.96, -1.11, -2.91,
-8.95, 14.29, -1.41],
 [-0.01, -54.86, 1.46, 0.03, -2.59, 4.47, 1.8, -0.76, 1.67, 0.34, 6.12, 3.34,
-139.21, -7.2],
 [-0.04, -57.08, -4.11, -1.84, 2.67, -1.82, -1.91, 0.16, -0.97, 0.85, -4.46,
2.98, -51.81, -30.28],
 [-0.0, -12.44, 2.95, 0.94, -3.64, -0.87, 4.8, 1.45, -0.29, 4.01, -5.57, 16.58,
-345.54, -87.28],
 [0.01, -0.16, -6.87, -3.71, -1.92, 1.89, -3.54, 3.94, -1.65, 7.43, -1.51,
-14.59, -251.62, -0.01],
 [-0.01, -0.02, -18.04, -7.59, -51.83, -86.17, -71.54, -107.24, -48.96, -95.22,
-2.52, -0.01, 0.01, -0.0]]
```


0
2
4
6
8
10
12
0
2
4
6
8
10
12




```
N=     1: score=    0.6967067467
N=     2: score=    0.2954618683
N=     3: score=    0.2270596313
N=     4: score=    0.1190390495
N=     5: score=    0.0540091239
N=     6: score=    0.0340005872
N=     7: score=    0.0185143469
N=     8: score=    0.0105583436
N=     9: score=    0.0059006104
N=    10: score=    0.0033308529
N=    11: score=    0.0018819415
N=    12: score=    0.0010696353
N=    13: score=    0.0006107751
```

```
N=    14: score=    0.0003504146
N=    15: score=    0.0002016547
N=    16: score=    0.0001161327
N=    17: score=    0.0000667174
N=    18: score=    0.0000381050
N=    19: score=    0.0000215689
N=    20: score=    0.0000120717
N=    21: score=    0.0000066713
N=    22: score=    0.0000036383
N=    23: score=    0.0000019582
N=    24: score=    0.0000010408
N=    25: score=    0.0000005468
N=    26: score=    0.0000002843
N=    27: score=    0.0000001466
N=    28: score=    0.0000000751
N=    29: score=    0.0000000383
N=    30: score=    0.0000000194
N=    31: score=    0.0000000098
N=    32: score=    0.0000000050
N=    33: score=    0.0000000025
N=    34: score=    0.0000000013
N=    35: score=    0.0000000006
N=    36: score=    0.0000000003
N=    37: score=    0.0000000002
N=    38: score=    0.0000000001
Optimization terminated successfully.
         Current function value: 0.000000
         Iterations: 37
         Function evaluations: 38
         Gradient evaluations: 38
```



12 Discussion

A few concluding remarks. As we have seen, a substantial fraction of ML applications are based on the idea of "probabilistically doing the right thing". This means that the question "how the training set was sampled" plays a very important role. Also, we leave it to the model to "discover", via training, what (correlation-)properties of input data might give clues that could be useful for the task at hand.

As a consequence, this "empirical risk minimization based" approach to "Artificial Intelligence" has a rather visible tendency to see and interpret the world that is quite different from how humans see the world.

Our current model architecture - and also the one we will discuss next - has an interesting property: the relative arrangement of pixels does not matter. So, if we decided on one particular way to

randomly reshuffle ("scramble") of all the 14×14, respectively 28×28 input pixels, and applied that same transformation to all training and test images, this would not impact classifier performance - whereas a human would be hopelessly lost with such a task.

Also, with this approach, there is a risk of models discovering that some particular "input data constellations" are "rarely seen on the training set, but tell-tale signs", so, "if these are present, this should contribute a lot of evidence that overshadows other signals". But then, if we use such a model on input data drawn from a different distribution than th one we drew training examples from, this can readily lead to mis-classifications where every actually intelligent interpreter of the data would wonder: "What the hell is going on?"

12.1 How things can go wrong

One illustrative such sample has been given in the paper "Can Your AI Differentiate Cats from Covid [12] - see Figure 1 from that publication.

Another similar such problem, observed in a model with text awareness, even made newspaper headlines such as with this article: `https://www.theguardian.com/technology/2021/mar/08/typographic-attack-pen-paper-fool-ai-thinking-apple-ipod-clip` - the underlying research is "Multimodal neurons in artificial neural networks [6]".

A highly relevant question for applications hence is: Can we understand how a particular classification produced by a ML model happened? Can we explore the justification for a given classification? For Deep Learning, the answer is pretty much always: No, except in quite unusual circumstances, where something very clever was done to allow this.

12.1.1 Adversarial attacks

Given that we know how to backpropagate gradients, we can of course also ask the question: Given a particular input, say an image, and a classification, how sensitive is the classification with respect to not model parameter changes (which is what we use in training), but input data changes - so, "what's the fastest way to distort this image of a penguin to make the model think this is a zebra?" Unfortunately, ML models can easily be sensitive to very low-amplitude correlations "which cannot be accidental", so just distorting an input image a little bit - if we have (say) gradients that give us some hint where to go - can utterly fool many a ML classifier.

The study of "adversarial attacks" on ML systems has a long history, with many inreresting publications. To quote from just one such paper, https://arxiv.org/abs/1412.6572 - Explaining and Harnessing Adversarial Examples [7]:

```
8 WHY DO ADVERSARIAL EXAMPLES GENERALIZE?

An intriguing aspect of adversarial examples is that an example
generated for one model is often misclassified by other models, even
when they have different architecures or were trained on disjoint
training sets. Moreover, when these different models misclassify an
adversarial example, they often agree with each other on its
class. Explanations based on extreme non-linearity and overfitting
cannot readily account for this behavior-why should multiple extremely
non-linear model with excess capacity consistently label
out-of-distribution points in the same way? This behavior is
```

```
especially surprising from the view of the hypothesis that adversarial
examples finely tile space like the rational numbers among the reals,
because in this view adversarial examples are common but occur only at
very precise locations.

Under the linear view, adversarial examples occur in broad
subspaces. The direction eta need only have positive dot product with
the gradient of the cost function, and epsilon need only be large
enough.  Fig. 4 demonstrates this phenomenon. By tracing out different
values of epsilon we see that adversarial examples occur in contiguous
regions of the 1-D subspace defined by the fast gradient sign method,
not in fine pockets. This explains why adversarial examples are
abundant and why an example misclassified by one classifier has a
fairly high prior probability of being misclassified by another
classifier.
```

12.1.2 Generative Models

Exploration of the phenomenon of adversarial attacks soon gave rise to the idea of pitching one ML model against another in a co-evolutionary "arms race" - with one model trying to generate plausible-looking random data (such as images trying to mimic portrait photos), and the other model trying to tell apart synthetic data from real data. While (co-)training such set-ups is a bit tricky, the results can be quite spectacular - see e.g. https://www.whichfaceisreal.com/ for a demo-and-game.

13 Appendices

13.1 Complex Backpropagation

Most ML practitioners only encounter the need to do complex-backpropagation when they try to extract signals from Fourier transformed audio data.

Naturally, situations where intermediate quantities are complex are much more common in theoretical physics, so it makes sense to spend some time on ensuring we understand how this works all the way down to the lowest level.

The code further up did incidentally use complex-backprop in one place (when working out coordinates of vertices along a ring on which a membrane ends), but this use case was non-essential, and we might easily have done this with real numbers only.

The general situation is: We have a real-valued objective function, but the calculation uses complex intermediate results.

It is still meaningful to change perspective and see each individual real/imaginary part as a real intermediate quantity that comes with its own sensitivity-accumulator. Also, it is still meaningful to ask: "by how much would the end result change if I intercepted the forward calculation and changed this intermediate quantity by ϵ"?

The open question then is: Can we lift all this back interpret these real-part-sensitivities and complex-part-sensitivities in terms of performing complex operations?

Let us look at "our final result is the imaginary part of a complex intermediate quantity", and "the calculation consists of multiplying the real input with `A+1j*B`":

```
def example(x):
  A, B = 10, 20j
  C = A + 1j * B
  result_c = C * x
  result = result_c.imag
  return result
```

If we instead wrote this as...:

```
def example_v2(x):
  A, B = 10, 20j
  result_c_re = A * x
  result_c_im = B * x
  return result_c_im
```

...it is immediately clear that our sensitivity is: `s_x = B * s_result_c_im`:

```
def example_v2_backprop(x):
  A, B = 10, 20j
  s_x = 0.0
  result_c_re = A * x
  result_c_im = B * x
  # return result_c_im
  s_result_c_im = 1
  s_x += B * s_result_c_im
```

...however, if, on the original code, we did `s_result_c = 0+1j; s_x += C * s_result_c`, this would be off - we would get `s_x = -B` rather than `s_x = B`. The answer is straightforward: If we regard complex numbers as $\mathbb{R}^2$-vectors, then complex sensitivities "live in the dual vector space", $\mathbb{R}^{2,*}$, and hence we need to complex-conjugate. This is basically the same story as with $dz = dx + i\,dy$ vs. $\frac{\partial}{\partial z} = \frac{\partial}{\partial x} - i\,\frac{\partial}{\partial y}$.

14 Basic ML Concepts and Ideas

14.1 Overview

1. More about our basic MNIST-themed binary classifiers
2. A Tour of General ML Concepts and Terminology

14.2 More about our MNIST classifiers

The simple "digit-8 recognizer" we built contains a few fundamental ideas that are ubiquitous, and hence useful to understand thoroughly - but we so far only justified them somewhat superficially as "plausible choices". Let us try to understand these more deeply.

The trainable classifier we built only has $14 \times 14 + 1$ model parameters - and rather limited performance. We will see later how to get better performance out of training models that can capture more complex high-dimensional structure via admitting nonlinearities.

Before we go there, let's also briefly come back to our "k nearest neighbors classifier" (For more background on this, see: https://en.wikipedia.org/wiki/K-nearest_neighbors_algorithm). While there are some choices to be made, such as about how many neighbors to take into account, how to scale the dimensions (actual features, or perhaps better feature-percentiles?), and how to weight neighbors, and we could use optimization to fine-tune these, we see that even with ad-hoc choices, this approach has an interesting property: If training set examples are drawn at random and independently from the same distribution as the examples we want to make predictions about, then, "given sufficiently many examples", "we ultimately could learn every structure that there is in high dimensional space". The number of examples needed to do this might be prohibitively large, but in principle, at some point, no matter what we see, we will have seen a few very similar training examples to use as a reference for our decision. Other approaches to learning then are, in a way about "using extra structural properties of the training examples in a clever way to be more resource-efficient in terms of how many examples we need". But as a first question when evaluating a ML application or proposal, it often makes sense to ask: "Could a k-nearest-neighbors classifier do this at least in principle, with very many training examples?"

The idea behind this approach resembles the design of "case law" - "make decisions in alignment with what was done on a closeby other example".

This clearly differs from human intelligence, and in ML, there is the technical term "transfer learning" (See e.g.: https://en.wikipedia.org/wiki/Transfer_learning) to describe a higher-level objective: Can a ML system generalize to examples beyond the training set distribution? For many architectures, the answer is "No" - a ML system may be able to learn to approximate the shape of a sine function over those 20 oscillations represented in the training set, but when asked to extrapolate beyond that range, would typically "just shoot off at random", demonstrating no "knowledge of the essence of an oscillation".

For practical applications, when using a k-nearest-neighbors approach, one often finds that using a fixed-but-essentially-random projection down to lower dimensions will retain much of the higher-dimensional probability density lumping/clustering properties while making the problem overall easier to manage.

Let us study those pieces of our optimization-based digit-8 classifier we have not discussed in detail yet "back-to-front", i.e. starting at the output side.

14.2.1 The Loss Function

For a given choice of parameters, we compute a score that somehow tells us "how well our classifier performs on the training set". Interestingly, we never actually use that score for anything relevant! We only use the gradient of that score-function to minimize, and a more primitive minimizing method, such as "iteratively taking a small step along the direction of the gradient" would work as well. (Subtle detail: The gradient is a co-vector, so "taking a step along the gradient direction" is a fishy concept if we lack a way to convert a co-vector to a vector, i.e. some scalar product. We will come back to this.) So, in a way, one important role of this quality-of-prediction-score is to ensure (via rot grad $\phi = 0$) "that the steps we take to improve performance won't make us run in circles". If we want to report the quality of the trained model, we generally would want to refer to other figures of merit - and in particular, since we want to use it for generalization, we want to report quality as measured on some test set.

But still - what does this quantity represent? If the label of a given training example y is 1 ("positive

case") or 0 ("negative case"), this example contributes $-y \log p - (1-y) \log (1-p)$ to the objective function we want to minimize, where p is our classifier's best estimate for "probability for this example to be a positive case".

One somewhat intuitive-and-also-quantitative way to think about this is based on a coding theory interpretation. As we will see, "this quantity measures the number of bits, in a proper continuous way, needed to encode the actual labels given the predictions of the classifier", or, to say the same thing in slightly different terms, "a quantitative measure (in bits) of the amount of correction needed to get from the classifier's output to the desired target 'message' ".

Let's approach this idea in small steps. Suppose we have an 'Alphabet' consisting of only 4 symbols, A, B, C, D (if you prefer, feel free to call these, for example, C, T, A, G). We want to communicate very long messages made of random uncorrelated sequences of these four letters. How many bits do we expect to need for a 100-letters message?

It actually depends on the probabilities with which each of these letters show up. If probabilities are 1/4-1/4-1/4-1/4, we can simply encode every letter with two bits, A=00, B=01, C=10, D=11, transport the message, and only need 200 bits. We can always do that, irrespective of probabilities, so we will never need more than 200 bits. But sometimes, we can do better. Obviously, "if C and D never show up", i.e. probabilities are 1/2/-1/2-0-0, we can simply encode A=0, B=1, and get away with 100 bits. Furthermore, "if no other letter than A ever shows up", there is "no surprise whatsoever", we know what a length-N message must look like, and we need 0 bits.

Now, how about a situation where probabilities are 1/2-1/4-1/4-0? Half the letters are A, and 1/4 are B, another 1/4 are C, and D never occurs? We have seen that we should use 1 bit to encode a p=1/2 letter, and 2 bits to encode a p=1/4 letter - and indeed, we can un-ambiguously decode a message that uses this encoding scheme: A=0, B=10, C=11. This uses $\sum_i p_i \log_{1/2} p_i = 0.5 \cdot 1 + 0.25 \cdot 2 + 0.25 \cdot 2 + 0 = 1.5$ bits per letter, so we would expect to need 150 bits for a 100-letter message. And we know we cannot do better than that. This is the idea behind "Huffman Coding". (The information content is called "entropy", and this is the same "entropy" as in statistical mechanics, and if the rumor is true that von Neumann proposed the term "entropy" somewhat jokingly, he must have been aware of this. "Number of different messages of that length, given this code" corresponds to "number of realizations of this ensemble" - and the identification carries over to the continuous case. The definitions only differ by a sign.)

Now, what do we do when probabilities do not align so nicely with powers of 2? Let's say they are 1/3-1/3-1/3-0? We could of course stick with the previous encoding scheme, requiring 1 bit for 1/3 of the letters, and 2 bits for the other cases, for a total of 5/3 bits/letter. There is a way to do better, though: Let us form words made of 5 letters: Each of them has probability $1/3^5 = 1/243$. With 8 bits, we can discriminate $2^8 = 256$ cases, so, attributing one 8-bit word per each of these 5-letter words, we even have a few codes to spare. (In a practical scheme, we might perhaps want to use of the extra 5 codes to express "end of message, and the previous word needs to be trimmed after 1,2,3,4,5 letters".) So, amortized, we get 8 bits / 5 letters = 1.6 bits/letter, which is somewhat better. Of course, we could look for even longer words, where $3^n/2^m$ comes closer to an integer (from below), such as encoding 41-letter words with 64-bit sequences for 1.585 bits/letter. The limit is, of course, $\log_2 3 \approx 1.5849625$ bits/letter.

Now, is there another perspective on this that readily generalizes to cases where probabilities are not nice fractions such as $1/2, 1/3, 1/4$? Given that we are talking about tunable models, we ideally would want to have a perspective where no probability is really special?

Such a perspective does indeed exist, and is known as "arithmetic coding" (see https://en.wikipedia.org/wiki/Arithmetic_coding). Technically speaking, this is not quite 100% true, since implementations of AC generally contain one deviation from the "pure" idea to make them practical, but basically, the idea is that we can map a long message that uses an alphabet with given letter-probabilities to a small sub-interval of the unit interval $[0, 1)$ as follows: We first partition this interval according to per-letter probabilities, in our 4-letter case, $[0, p_A), [p_A, p_A + p_B), [p_A + p_B, p_A + p_B + p_C), [p_A + p_B + p_C, 1)$. The first letter selects one of these intervals. We then partition the selected interval with the same proportions, and use the 2nd letter to select a sub-interval, and-so-on. At the end of our message, long as it may be, we have a small interval remaining that contains all real numbers which we could take as valid representations of this message. If b is an integer such that $1/2^b$ is smaller than the length of this interval, then this final interval is guaranteed to contain an integer multiple m of $1/2^b$, with $0 \leq m < 2^b$, and this way, we can encode our message with b bits. In the limit of very long messages, we need $\log_{1/2} p_X$ bits to encode a letter X with probability p.

So, if sender and receiver have a shared collection of to-be-classified examples, and access to the same (trained) classifier, and the sender also knows the "ground truth" classifications and wants to transmit that information via a compactly encoded message, it would only need to send corrections/amendments to the classifier's output. Let us study a hypothetical case. We encode a "Yes" as "picking the left part of a subdivided interval".

- Example 0 has ground truth label "Yes", and shared classifier says: "p(Yes) = 80%".

  Sender narrows the [0, 1)-interval to [0.0, 0.8).

- Example 1 has ground truth label "No", and shared classifier says: "p(Yes) = 50%" (i.e. classifier says "I have a hard time saying anything here").

  Sender narrows the [0, 0.8)-interval to [0.4, 0.8).

- Example 2 has ground truth label "No", and shared classifier is rather confident, saying "p(Yes) = 0.1%".

  Sender narrows the [0.4, 0.8)-interval by shaving off 0.1% of the range at the lower end, to [0.4004, 0.8).

- Example 3 has ground truth label "Yes", but the shared classifier somewhat confidently thinks otherwise, telling us "p(Yes) = 10%".

  Sender narrows the [0.4004, 0.8) range down to [0.4004, 0.44036).

- Example 4 has ground truth label "No", and the shared classifier is leaning towards a "Yes", telling us "$p(Yes) = 75\%$"

  Sender narrows the [0.428372, 0.44036) range down to [0.43037, 0.44036)

- We are out of examples. The sender sees that the final interval has length $\approx 1/100.1$ and sends the 7-bit binary number (most significant bit first) 111000, which represents 56/128=0.4375 and lies in the final interval.

- Receiver classifies example 0, gets "p(Yes) = 80%" (like sender), sees "The number 0.4375 is in the [0, 0.8) sub-interval", infers "label=Yes", narrows the interval to [0.0, 0.8).

- Receiver classifies example 1, gets "p(Yes) = 50%", sees "The number 0.4375 is in the right half of the 50-50 split of the remaining interval", infers "label=No", and narrows the interval

to [0.4, 0.8).

- ... and-so-on.

Nicely, we can directly convert the length of the remaining interval to a number of bits by a continuous and smooth function: $b = \log_{1/2} p$. Adding a letter shrinks interval-length multiplicatively, but affects the logarithm additively, so it makes sense to talk about "expectation value of the logarithm of the final interval's length" - this then is the "alphabet(/code)'s entropy times message length".

Now, what happens if we use "number of bits (as measured in the arithmetic-encoding way)" - if you want so: a quantitative measure of the "surprise" - to measure quality of the predictions of our ML model? (The overall factor to convert between $\log_2$ and $\log_e$ is if course irrelevant for minimization.) At first, the largest contributions may well come from cases where the classifier accidentally expresses confidence but is off. Here, gradients pull it towards "I should not be so sure about this case". Fine-tuning a "I can even be more certain about this case" situations contributes less - if a classifier can go from "99% certain about this case" to "99.9% certain" and is right with its assessment, this merely changes the interval-length by a factor of about 1.0091, removing 0.013 bits of "surprise".

Conceptually, this "loss function", which is called "cross-entropy" is some sort of "distance between probability distributions" (but a non-symmetric distance, $d(A, B) \neq d(B, A)$ in general - we will come back to it). The same construction is also used to answer the question "how many bits extra do we expect to expend when encoding a random-letters message where the letters follow probability distribution A using a coding scheme as if they followed probability distribution B" - and in this context, it is called "relative entropy" or also "Kullback-Leibler divergence" (see: https://en.wikipedia.org/wiki/Kullback%E2%80%93Leibler_divergence).

Naturally, we expect properly set up and trained models to be "calibrated" in the sense that if we take 1000 test examples for which a binary classifier gives us a probability for the example to carry a positive label in the range 60%-61% to have a positive-examples fraction of 60%-61% (of course then also with sampling noise). It might be that our first classifier is too simple to behave that way, but calibration is in general a useful canary to check that can flag up problems early.

Unfortunately, with the advent of "deep learning" architectures and the focus-shift to "better performance on simple metrics such as classification accuracy makes work publishable", proper model calibration is no longer what one can usually expect.

See Figure 1 of https://arxiv.org/pdf/1706.04599.pdf - On Calibration of Modern Neural Networks [9].

14.2.2 The Logistic ("sigmoid") Function

The other question is why we use the "sigmoid function" $x \mapsto \frac{1}{1+\exp(b-x)}$ (effectively the Fermi-Dirac distribution function $\mu \mapsto \frac{1}{1+\exp(\beta(E-\mu))}$) to obtain probabilities.

At a simplistic level, "we want to map evidence to probability, and probabilities are in the range [0, 1]", so we should use a function with range such as $\mathbb{R} \to (0, 1)$, and the sigmoid fits that bill. There is however a deeper reason.

A nice example that provides some useful background here is email SPAM filtering. (SPAM = unsolicited commercial email.) While the questions "why this problem exists at all" and also "whether automated spam filtering actually is a meaningful approach" very much are worthwhile to discuss,

we want to concern us with a different aspect: If we wanted to do automated spam detection - how could one build a useful classifier?

Until about 2002, email SPAM filtering was dominated by hand-crafted and curated spam detection rules. This changed in 2002 with an online article by Lisp Hacker and writer Paul Graham (http://www.paulgraham.com/spam.html - A Plan for Spam [8]). In hindsight, the reason this article exists is that the author was not aware of earlier - flawed - research that looked into this approach and found it to not work. The gist of Paul Graham's idea is that we should try a "naive Bayesian" approach, look for simple signals in favor and also against a given email message to be SPAM, and then combine these signals into a spam-probability. Interestingly, in 2002, the rather crude model proposed by Paul Graham did indeed work very well - so well indeed that it was widely adopted and led to a wave of very weird-looking SPAM that tried to trick classifiers based on this idea. But what, then, was the idea? Basically:

1. If we have a somewhat large corpus of personal email messages labeled as "SPAM", respectively "not-SPAM", it is simple to build a lexicon out of the dictionary words in all these messages and estimate, per-word, what the likelihood is for an email containing this word to be SPAM. (Different people's mailbox may give very different-looking such dictionaries. The term "supergravity" is generally not present in most people's mailboxes, and where it is, it generally is strong "not SPAM" evidence - while the term "prince" may well provide SPAM evidence for most people.)

2. We bluntly treat all SPAM-relevant words as "independent experts". Suppose an email message contains only two SPAM-relevant terms, A and B, and we have "p_A={probability estimate for an email message containing the term A to be SPAM}," as well as "p_B ={probability estimate for an email message containing the term B to be SPAM}". What is then a good probability estimate for the email to be SPAM if these simple-to-compute "experts" were independent (which, of course, is a very crude and doubtful assumption).

Let's say we have a large body of email messages that have been sampled in such a way that half of them are SPAM. A small fraction q_A of these messages contain the term A, and an independent small fraction q_B of these messages contain the term B. We determine p_A and p_B as above. Now, on the fraction $q_A \cdot q_B$ of email messages that contain both the term A and B, some are SPAM and some are not SPAM. The relative sizes of these fractions are $p_A \cdot p_B$ and $(1-p_A)\cdot(1-p_B)$, so the SPAM probability among these messages is $p_{AB} = \frac{p_A p_B}{p_A p_B + (1-p_A)(1-p_B)}$. So, we have the following binary function `$C:{\mathbb R}\times{\mathbb R}\to{\mathbb R}$` for combining probabilities:

$$C(a,b) = \frac{a \cdot b}{a \cdot b + (1-a)\cdot(1-b)}.$$

Naturally, we have $C(b,a) = C(a,b)$, as well as $C(a,0.5) = a$ and also $C(C(a,b),c) = C(a,C(b,c))$ - and as we would expect from this motivation, a bit of algebra gives us $C(C(a,b),c) = \frac{a\cdot b\cdot c}{a\cdot b\cdot c+(1-a)\cdot(1-b)\cdot(1-c)}$.

Now, this structurally resembles a situation we know well from special relativity: the relativistically correct formula for adding collinear velocities (using $c = 1$) is $s(v,w) = \frac{v+w}{1+vw}$, and we also have commutativity, associativity, and a neutral element. For relativistic velocities, we know that the story simplifies a lot if we regard velocities as "dressed up other quantities" which just can be added linearly. These "other quantities" we call "rapidities", and the dressing/undressing is done by $u = \tanh^{-1} v$, respectively $v = \tanh u$.

So, is there a corresponding pair of a dressing/undressing function that trivializes our "addition function" $C(a,b)$ above - making `C(.,.)` merely dressing-up a sum after un-dressing the summands? Our neutral element is $p_A = 0.5$, and this should correspond to "evidence zero". (Clearly, adding another independent expert who merely flips a fair coin does not help or hurt us with any classification task.) Let us call our (not yet known) evidence-to-probability function $P : \mathbb{R} \to (0,1)$. We know that $P(0) = 0.5$ holds. Also, we expect P to be smooth. Let us define $P'(0) := \gamma$, so that we have an expansion $P(\epsilon) = 0.5 + \gamma\epsilon + \mathcal{O}(\epsilon^2)$. Since we want $P(x+y) = P(x)P(y)/(P(x)P(y)+(1-P(x))(1-P(y))$ to hold, we can use this to expand $P(x+\epsilon)$ to 1st order in ϵ and derive the differential equation $P'(x) = 4\gamma P(x)(1-P(x))$.

So far, γ is an arbitrary slope-paramter. We naturally would want $\gamma < 0$ ("more positive evidence increases the probability"), but its value basically is merely about picking some nice normalization on the evidence-scale (like "setting $c = 1$" for special relativity). Here, a natural choice is $\gamma = 1/4$, and with this we get the ODE $P' = P(1-P)$.

The next steps are straightforward: Let us define $Q(x) := 1/P(x)$, which gives us $Q'/(Q-1) = -1$, so $\log(Q(x)-1) = -x + c$, and from this we find (for $\gamma = 1/4$, and also using $P(0) = 0.5$):

$$P(x) = \frac{1}{1+\exp(-x)}$$

.

In the context of statistics (or ML), this is called the "logistic" function. Its inverse is the "logit" function:

$$P^{-1}(p) = \log\frac{p}{1-p} = 2\tanh^{-1}(2p-1)$$

.

14.2.3 Multiclass classification

While we now fully understand every aspect of the construction of our simple digit-classifier, there is one detail left to look into: So far, we discussed a simplified "binary classification" setting. How about 1-in-10 classification? We could of course train 10 binary classifiers ("this vs. everything else"), and then somehow try to merge these. In terms of fundamental properties, what would we expect the "loss function" to look like? Important desiderata are:

- We still want to be able to map some "(linearly) accumulated independent (per-class) evidence" to (per-class) probability.
- Using relative entropy / KL divergence as the objective function still seems to make sense. For the "correct" label, we get a contribution of $-\log p_i$, and for each incorrect label, $-\log(1-p_k)$.
- We again would like to have some simple "naive Bayesian" interpretation to hold. To give a simple example, for two independent 1-out-of-3 classifiers that see evidence-vectors (of independent evidence) (e_{1A}, e_{1B}, e_{1C}), respectively (e_{2A}, e_{2B}, e_{2C}), and attribute to these the corresponding probability-vectors (p_{1A}, p_{1B}, p_{1C}), (p_{2A}, p_{2B}, p_{2C}), we would like identities such as this one to hold:

$$p_{(1+2)A} = \frac{p_{1A}p_{2A}}{p_{1A}p_{2A} + p_{1B}p_{2B} + p_{1C}p_{2C}}$$

In words: combining the assessments from both independent experts, out of all examples that expert 1 attributes probabilities (p_{1A}, p_{1B}, p_{1C}) to, and expert 2 attributes probabilities (p_{2A}, p_{2B}, p_{2C}) to [fine print: where these probabilities are representative for some chunk of probability-space], the relative odds for "true label is A" : "true label is B" are $p_{1A}p_{2A} : p_{1B}p_{2B}$, and likewise for the other two combinations.

If we want probabilities to behave multiplicatively when we add independent evidence, this means that the probability-ratio for two classes $p_A : p_B$ should be given by the exponential of the evidence difference: $p_A : p_B = \exp(e_A - e_B)$. This then gives us these probabilities:

$$(p_A, p_B, p_C) = (\exp(e_A)/Z, \exp(e_B)/Z, \exp(e_C)/Z), \quad \text{with } Z := \exp(e_A) + \exp(e_B) + \exp(e_C).$$

Or, for N classes:

$$p_i = \frac{\exp e_i}{\sum_k \exp e_k}.$$

[**Participant Exercise**: Show that in the case of binary classification, this indeed reduces to the logistic function.]

This function, called "softmax" in a Machine Learning context, is of course just the Boltzmann distribution. Intuitively, there is little surprise to the physics-trained eye here (perhaps depending on how much of a physical chemist one is). If we think of an individual example's total classification-probability as some kind of molecular substance where each individual molecule can be in one out of multiple discriminable but energetically-equivalent configurations (each configuration corresponding to a ML label class), then the thermodynamic-equilibrium proportions of the different configurations (ML perspective: per-class probabilities for the example) would be determined by the chemical potential. In this picture, the accumulated per-class-evidence plays the role of a "chemical potential" (up to a sign).

14.2.4 More about the ("pedestrian"-)thermodynamics interpretation

Suppose we build an image classifier that can discriminate three types of fruit: apples, oranges, and bananas. Let's say that for each type of fruit, there is a hidden "internal" classification system that an expert on the fruit could use to discern three independent describing attributes, such as "variety" (with, say, 5 options per fruit-type), "ripeness" (typically going from "still green" to "green once again" - let's say also with 5 options), and "imperfections" (also with 5 options).

Now, let's say the probability for each combination of `(fruit-type, fruit-variety, fruit-ripeness, fruit-imperfections)` is the same according to the distribution from which we draw our examples.

What happens if our ML-classifier, when analyzing an example, extracted a feature that could be described in approximate-human-terms as "this fruit has a brown spot"? Let's say two of the five imperfection-categories for bananas and apples are compatible with "brown spot observed", but only one for "oranges". Then, this one feature-observation opens up a one-of-two-choices degree of freedom for apples and bananas, but we only have a one-of-five-choice for oranges. Then, there are twice as many ways how the observations could be compatible with "we are looking at an apple" in comparison to "we are looking at an orange". Likewise for banana-vs-orange. So, the "space of

possibilities" has acquired a relative factor 2 each for two fruit types, and "evidence", which should be a linear additive quantity measuring the space of options (like entropy) hence should increase by the log of that factor. Knowing nothing else, having observed a brown spot overall makes it less likely that we are looking at an orange, but this is of course a pure gambler's argument that does not involve any "understanding" whatsoever.

Now, let us compare this to a pedagogically-constructed chemical situation. Suppose we have a gas of 5-atomic molecules, a carbon-12 "stereocenter" in the middle and four different atoms around it - let's say F, Cl, Br, I. Each molecule can be in left-handed ("S") or right-handed ("R") configuration. Let's say we also provide a catalyst that allows removing and reattaching an I-atom, and when reacting with the catalyst, the left-handed form can turn into the right-handed form and vice versa. Furthermore, let's assume each of these odd-number-of-protons nuclei had a total nuclear spin of 1/2. Then, focusing on nuclear spin, we could describe that part of the quantum state of such a molecule as an element of a 2x2x2x2-dimensional Hilbert space, one dimension per nucleus. If we started from 100% left-handed molecules, interaction with the catalyst would get us into an equilibrium that is a 50%-left-50%-right mixture. Now, suppose we had some "thermodynamic daemon" type magical catalyst which, whenever a left-handed molecule gets turned into a right-handed one, it replaced the iodine atom with one that has nuclear spin zero, and replaced the spin-zero iodine on a right-handed molecule with a spin-1/2 one as it gets turned into a left-handed molecule again? Then, the nuclear-quantum-state Hilbert space for the right handed form would be 2x2x2x1-dimensional, "having one degree of freedom less". In terms of "logarithm of the size of the space of opportunities", there is a log(2)-difference between these cases, and this then is the contribution to the chemical potential that favors the "more configurations available" left handed form. Going back from chemical potential to probabilities, we exponentiate that difference again, and get a 2/3-left-to-1/3-right ratio, which we can attribute to "left form having one degree of freedom more".

15 Some Concepts and Terminology

We built a rather simplistic "logistic regression based" handwritten digit classifier. Before we explore how to improve this, we should look into some concepts and terminology that allow us to form an idea of what "classifier A performs better than classifier B" means.

For a simple benchmark problem such as "handwritten digit classification" where each digit is equally likely1, a very basic performance metric, measured on some "test set" that was set aside so that it could not possibly have affected decisions entering the design or training of the classifier, is **accuracy**, i.e. the fraction of correctly classified examples. This, however, can be misleading, especially...

- ...if some mis-classifications are substantially more painful than others, and also
- ...if there is substantial class imbalance, such as: "99% of examples are Negative anyhow".

Let us first focus on some terms related to **binary classification** tasks. The difference between "type I error" and "type II error" is commonly discussed in secondary education: A "type I error" is a "False Positive" - "we cry wolf but there is no wolf". A "type II error" is a "False Negative" - "we fail to cry wolf but should have". Overall, we have:

- True Positive Fraction (TP) - fraction of true positives (ground truth label = yes; classification = yes) in the example set.

- True Negative Fraction (TN) - fraction of label=no, classification=no.
- False Positive Fraction (FP) - fraction of label=no, classification=yes.
- False Negative Fraction (FN) - fraction of label=yes, classification=no.

In terms of these, "accuracy" is TP+TN, i.e. the fraction of correctly classified cases. We also define "recall" as "fraction of label=yes examples we manage to classify as positive", i.e. TP/(TP+FN), as well as "precision" as "fraction of label=yes examples among those we classify as positive", i.e. TP/(TP+FP). There are other - derived - metrics such as F1-score, which are however less commonly reported.

In general, for binary classification tasks, our classifier will produce some sort of "confidence" for a classification, and with proper calibration, this may take the form of "I am 80% confident this is a Yes". We would then expect about 20% of all those example to which a (properly calibrated!) classifier attributes a probability in the interval $[80\% - \epsilon, 80\% + \epsilon]$ to actually be negative cases.

We obviously have a choice of a threshold here: Do we want to play it safe and only classify examples as "yes" where the classifier is $\geq 99\%$ certain of that classification, accepting that we will miss some positive cases, or do we want to catch as much as we can, classifying even "slightly suspicious cases" as "interesting (such as: for subsequent investigation)" as "positive", perhaps using even a threshold of only 1% - or something in between?

Let us consider a fixed set of labeled examples and see what happens if we start from a threshold of 100% and gradually lower it.

Each example has been given a score by the classifier, and all that matters is the ordering of examples by score, and what happens when lowering the threshold crosses the score of the next example. These properties are invariant under mapping classifier score to some other score as long as we use a strictly monotonically increasing mapping.

For thresholds higher than the maximal example score, we classify nothing as positive, so have TP = FP = 0.

For thresholds lower than the minimal example score, we classify everything as positive, and have TP = FP = 1.

What happens if we start at high threshold, TP=FP=0 and keep lowering the threshold? We keep crossing examples which change their classification from "negative" to "positive". If the label was "positive", this increases TP by 1/{number of positive examples}. If the label was "negative", this increases FP by 1/{number of negative examples}. So, we either take a small step in +x or in +y direction on the (x=FP, y=TP) plot.

If the "classifier" just randomly guessed scores, and we have a fraction p of positive examples, then at every score-crossing, we will, with probability p, take a step of size 1/p in y=TP direction, and with probability 1-p take a step of size 1/(1-p) in x=FP direction. So, the TP/FP-for-all-thresholds curve will be a "noisy diagonal line". One relevant metric is ROC-AUC, the "receiver operating characteristic area-under-curve". Here, the TP/FP-curve just described is called the "receiver operating characteristic" (ROC). This is a threshold-invariant measure of classifier performance. For a good-quality classifier, we expect that lowering the threshold starts catching many correctly-classified examples, so TP should go up with FP ideally only increasing a little - while for low thresholds, we ultimately also mis-classify all negative examples as positive, and the curve reaches (x=TP=1, y=FP=1).

The ROC-AUC can be interpreted directly as the probability for a randomly picked positive-label

and negative-label example to have classifier-scores "in good order", i.e. positive-label example has higher score than negative-label example. This is so for the following reason: If we lower the threshold until we make the next False-Positive mis-classification, we move a stretch of 1/(1-p) along the x=FPR axis, at y=TPR-at-this-new-threshold. So, to each negative example, we can attribute a rectangle on the graph that measures the fraction of positive examples among all positive examples which have a score lower than this one.

Details and plots can be found on: https://en.wikipedia.org/wiki/Receiver_operating_characteristic.

15.1 A DL Classifier "from scratch"

Let us see if we can build a full-fledged "Deep" Neural Network based handwritten digit classifier entirely from scratch (only a few hidden layers, mostly to illustrate the key ideas).

This is indeed feasible, but we need to look into a few tricks.

Pre-DL, the situation was that, even if we managed to train a fully-connected deep neural network with more than 2, perhaps 3, layers, this generally would not have managed to add any benefit. Nowadays, we know how to even get performance improvements from extra layers at 100+ layers deep.

Why do this "from scratch"? As physicists, "we understand what we can create", and we may sometimes find ourselves in a situation where we want to make changes to something that just about every ML framework would not allow us to change.

What is the basic idea behind "deep learning" architectures? So far, we have built a simple logistic regression classifier, "where every pixel was an input feature". We could consider improving this by adding "feature products", i.e. also add tunable weights for products-of-intensities-of-two-pixels. We might want to simplify this by first performing a (tuneable) linear projection from $N \times N$ pixels to some D-dimensional space, $D \ll N \times N$, and then taking our features from perhaps $D \oplus D \otimes D \oplus D \otimes D \otimes D \oplus \ldots$ - and this would indeed "allow learning to become sensitive to more complicated high-dimensional distribution structure".

Depending on the problem, we might even hand-craft some of the features, perhaps after doing extensive research on "where the evidence sits". Such "feature engineering" was an important part of pre-DL ML, and a common situation was that ML turned out to be "merely a tool to guide problem analysis, and once we understand the problem, it falls apart and we see how to do this without using ML".

Post-DL-revolution, the question of which model architectures (/ architecture elements or ideas) are especially useful for which types of problems is still relevant, but the focus has shifted away from engineering extraction of individual features. Often, we are in a situation where "the entire tree gets pushed in on one side, and a table comes out on the other side, and we can leave it to ML training to figure out on its own what part is useful to make what part of the result."

For handwritten digit classification, making this possible is mostly about avoiding some earlier mistakes.

The basic architectural choice for this exercise will be a rather simple fully connected Neural Network, where each layer has a certain number of units each of which receives a trainable linear combination of the outputs of the previous layer as input, and applies a simple nonlinear-but-differentiable

$\mathbb{R} \to \mathbb{R}$ function to this linear combination to produce its output.

Specifically, let us use these tricks (there are a few more which we will, for now, deliberately ignore):

- We normalize our input-vectors such that features have mean zero and variance 1.
- We use many training examples (tens of thousands).
- We use a blunt approach to fast training ("stochastic gradient descent"):
  - Estimate, rather than compute, the training set loss (rather, its gradient) by only considering a small random sample of all training set exercises (such as: 32) for each parameter-update.
  - Multiply the ∂(loss)/∂(parameter k)-gradient with a small (negative) step-size, the "learning rate", and update parameters by walking in that direction.
- We pay some attention to random parameter-initialization, the basic idea being: "If for every input, activations are roughly normal-distributed, this should also hold for every output - if inputs are independent."

This then leads us to the code shown below - here, for the last time, with hand-crafted backpropagation.

For this example, we will need some nonlinearity. The actual choice does not matter too much, but let us pick a function that is symmetric around zero, has a near-zero linear region where it is approximated well by the identity, stays very roughly linear for a value of up to about 1 or so, has a somewhat simple analytic derivative, and does not flatten exponentially, so further-out regions still do contribute somewhat to gradients. One function that fits the bill here is the inverse tangent (arctangent) function.

[ ]:
```
# Importing the relevant Python libraries ("modules").
import math
import pprint
import time

import matplotlib
from matplotlib import pyplot
import numpy
import scipy.optimize
import tensorflow as tf

# Larger figures.
matplotlib.rcParams['figure.figsize'] = (20, 10)
```

[ ]:
```
def get_nn_classifier_func(parameters=()):
  """Returns a neural network classifier function.

  Args:
    parameters: A sequence of 2-index numpy.ndarray-s `wb`, each of shape
      [layer_width + 1, next_layer_width] that represent the weights
      (indexed `wb[input_node_index, output_node_index]`) and biases
      (indexed `wb[layer_width, output_node_index]`) for the inter-layer
      connections.
```

```
  Returns:
    A function `f` with signature
    ```def f(batch_features, batch_labels, want_gradient=False):
         ...
         return loss, softmax_probs, opt_grad_param_pieces
    ```
    where `batch_features` is a [batch_size, num_features] numpy-array-like
    value providing batch-features (which will internally get converted
    to a numpy array), `batch_labels` is a [batch_size, num_classes]
    numpy-array-like value providing per-example labels in one-hot encoded
    form, batch-indexed in parallel to `batch_features`, and `want_gradient`
    determines whether gradients should be computed. If this is logically
    false, the `opt_grad_param_pieces` part of the returned tuple is None,
    otherwise it is a sequence of numpy-arrays indexed in parallel to
    `parameters` which provides the corresponding gradient-component
    of the loss with respect to the parameters. Correspondingly, the `loss`
    part of the returned tuple is the averaged-across-batch-examples
    cross-entropy loss of the prediction vs. the given labels on the batch,
    and `softmax_probs` is a vector of predicted probabilities.
  """
  def loss_pred_grad(batch_features, batch_labels,
                     want_gradient=False):
    """Runs inference and optionally computes gradients."""
    # Labels must be one-hot!
    batch_features = numpy.asarray(batch_features)
    batch_labels = numpy.asarray(batch_labels, dtype=batch_features.dtype)
    intermediate_data = []
    batch_size = batch_features.shape[0]
    num_classes = batch_labels.shape[1]
    # Per batch-example, turn input features into a vector.
    activations_now = batch_features.reshape(batch_features.shape[0], -1)
    for num_layer, layer_params in enumerate(parameters):
      weights = layer_params[:-1, :]
      biases = layer_params[-1:, :]
      pre_nonlinearity = numpy.einsum('bi,io->bo',
                                      activations_now,
                                      weights) + biases
      if num_layer + 1 != len(parameters):
        post_nonlinearity = numpy.arctan(pre_nonlinearity)
      else:
        # No activation on final layer, which inputs into softmax.
        post_nonlinearity = pre_nonlinearity
      intermediate_data.append((activations_now,
                                pre_nonlinearity,
                                post_nonlinearity))
      activations_now = post_nonlinearity
```

```
    # We still have to convert the final activations to probabilities --
    # which we do via softmax. We use a trick to avoid large arguments
    # to exp(), shifting all activations to max=0.
    max_activation_indices = activations_now.argmax(axis=-1)
    max_activations = numpy.fromiter(
        (example[k] for example, k in zip(activations_now,␣
↪max_activation_indices)),
        dtype=activations_now.dtype)[:, numpy.newaxis]
    max_activations = activations_now.max(axis=-1, keepdims=True)
    shifted_activations = activations_now - max_activations
    softmax_weights = numpy.exp(shifted_activations)
    softmax_denoms = softmax_weights.sum(axis=-1, keepdims=True)
    softmax_probs = softmax_weights / softmax_denoms
    entropy_contribs = numpy.where(batch_labels,
                                   -numpy.log(1e-12+softmax_probs),
                                   -numpy.log(1-softmax_probs+1e-12))
    loss = entropy_contribs.sum(axis=-1).mean()  # per-batch-example
    #
    if not want_gradient:
      return loss, softmax_probs, None
    # We want a gradient.
    # Backpropagation code below.
    # [We already know how this works, so this may perhaps be only be of
    #  interest for participants who want to see more code examples. We need
    #  not discuss this in detail. "No really new tricks here."]
    s_entropy_contribs = numpy.full([batch_size, num_classes], 1 / batch_size)
    s_log_softmax_probs = numpy.where(batch_labels,
                                      -s_entropy_contribs, 0)
    s_log_complementary_softmax_probs = numpy.where(batch_labels, 0,
                                                    -s_entropy_contribs)
    # Here, we recompute some simple intermediate quantities which we did not
    # bother to hold on to.
    s_softmax_probs = s_log_softmax_probs / (1e-12 + softmax_probs)
    s_softmax_probs -= s_log_complementary_softmax_probs /␣
↪(1-softmax_probs+1e-12)
    s_softmax_denoms = -(s_softmax_probs * softmax_weights /
                         softmax_denoms**2).sum(axis=-1, keepdims=True)
    s_softmax_weights = s_softmax_probs / softmax_denoms
    s_softmax_weights += s_softmax_denoms
    s_shifted_activations = softmax_weights * s_softmax_weights
    s_max_activations = -s_shifted_activations.sum(axis=-1)
    s_activations_now = s_shifted_activations
    # This step is a bit messy - we here have to do in Python what ought
    # to be handled in NumPy.
    for num_example, k in enumerate(max_activation_indices):
      s_activations_now[num_example, k] += s_max_activations[k]
    # Here, `s_activations_now` are the sensitivities on the final inputs
```

```
    # to softmax.
    grad_param_pieces = [None] * len(parameters)
    for num_layer_transition in reversed(range(len(parameters))):
      k = num_layer_transition # Abbrev.
      s_post_nonlinearity = s_activations_now
      activations, pre_nonlinearity, post_nonlinearity = (
          intermediate_data[num_layer_transition])
      if num_layer_transition + 1 != len(parameters):
        s_pre_nonlinearity = s_post_nonlinearity / (
            1 + numpy.square(pre_nonlinearity))
      else:
        s_pre_nonlinearity = s_post_nonlinearity
      # Forward code was:
      # weights = layer_params[:-1, :]
      # biases = layer_params[-1:, :]
      # pre_nonlinearity = numpy.einsum('bi,io->bo',
      #                                 activations_now,
      #                                 weights) + biases
      layer_params = parameters[num_layer_transition]
      weights = layer_params[:-1, :]
      biases = layer_params[-1, :]
      s_layer_params = numpy.zeros_like(layer_params)
      # Bias-sensitivities.
      s_layer_params[-1, :] += s_pre_nonlinearity.sum(axis=0)
      s_layer_params[:-1, :] += numpy.einsum(
          'bo,bi->io', s_pre_nonlinearity, activations)
      s_activations_now = numpy.einsum(
          'bo,io->bi', s_pre_nonlinearity, weights)
      grad_param_pieces[num_layer_transition] = s_layer_params
    return loss, softmax_probs, grad_param_pieces
  #
  return loss_pred_grad
```

[ ]:
```
# Let us first try out a really simple "random" example,
# checking if our gradients work. (They do!)
rng = numpy.random.RandomState(seed=0)
demo_params = [rng.normal(size=(2+1, 5), scale=0.4),
               rng.normal(size=(5+1, 4), scale=0.4),
               rng.normal(size=(4+1, 2), scale=0.4)]
demo_batch_features, demo_batch_labels = (
    rng.normal(size=(3, 2), scale=0.1),
    numpy.array([[0, 1], [0, 1], [1, 0]]))

f_demo = get_nn_classifier_func(demo_params)
f_demo_val, _, f_demo_grad = f_demo(demo_batch_features, demo_batch_labels,
                                    want_gradient=True)
print('f_demo_val:', f_demo_val.round(6).tolist())
```

```
pprint.pprint([t.round(5).tolist() for t in f_demo_grad])

def get_numerical_gradient(n, idx1, idx2, eps=1e-6):
  # We are mutating params[n][idx1, idx2] by eps and evaluate the difference.
  demo_params_eps = list(demo_params)  # Copy.
  demo_params_eps[n] = demo_params_eps[n].copy()
  demo_params_eps[n][idx1, idx2] += eps
  f_demo_eps = get_nn_classifier_func(demo_params_eps)
  f_demo_eps_val, _, _ = f_demo_eps(demo_batch_features,
                                    demo_batch_labels,
                                    want_gradient=True)
  print('f_demo_eps_val:', f_demo_eps_val, 'vs.', f_demo_val)
  return (f_demo_eps_val - f_demo_val) / eps

print('###')
print(get_numerical_gradient(0, 0, -1))
```

```
f_demo_val: 2.175762
[[[0.0019, 0.0026, -0.01194, 0.01522, -0.01891],
  [-0.00507, -0.00759, 0.03308, -0.04231, 0.05237],
  [0.06119, 0.09462, -0.40677, 0.5206, -0.64192]],
 [[0.02846, 0.01043, 0.087, 0.03194],
  [0.1313, 0.04752, 0.4027, 0.14725],
  [0.08044, 0.02917, 0.24657, 0.09022],
  [0.02027, 0.00746, 0.06195, 0.02275],
  [0.04768, 0.01734, 0.14611, 0.05349],
  [0.25897, 0.09375, 0.79419, 0.29044]],
 [[-0.14684, 0.14684],
  [0.26458, -0.26458],
  [0.27931, -0.27931],
  [0.01699, -0.01699],
  [0.89155, -0.89155]]]
###
f_demo_eps_val: 2.1757617947269394 vs. 2.175761813641268
-0.018914328769881195
```

[ ]:
```
# Let us get the MNIST training set...

import tensorflow_datasets as tfds

(ds_train, ds_test), ds_info = tfds.load(
    'mnist',
    split=['train', 'test'],
    shuffle_files=False,
    as_supervised=True,
    with_info=True,
)
```

```
images_and_labels_all = list(ds_train.as_numpy_iterator())
print('Number of MNIST training-set examples:', len(images_and_labels_all))
# We build our training and test set from the MNIST "training" examples.
# (If we used the `test` examples in any way here, we would risk making
# decisions about our model based on official test set performance,
# hence end up "training on the for-evaluation test set". We must avoid this!)
#
# ...but let us shuffle them randomly-but-reproducibly.
rng = numpy.random.RandomState(seed=0)
rng.shuffle(images_and_labels_all)
num_training = 4 * len(images_and_labels_all) // 5  # 80% training examples.
images_and_labels_training = images_and_labels_all[:num_training]
images_and_labels_ourtest = images_and_labels_all[num_training:]
print('\n######\n')
print('Number of examples in our training set:', num_training)
```

```
Number of MNIST training-set examples: 60000

######

Number of examples in our training set: 48000
```

[ ]:
```
# We need to map the labels to one-hot form.
labels_1hot_all = numpy.zeros([len(images_and_labels_all), 10])
labels_1hot_all[numpy.arange(len(images_and_labels_all)),
                numpy.fromiter(
                    (label for _, label in images_and_labels_all),
                    dtype=numpy.int32)] = 1.0
print('### Labels')
print(labels_1hot_all[:5])

labels_1hot_training = labels_1hot_all[:num_training]
labels_1hot_ourtest = labels_1hot_all[num_training:]

img0_all = numpy.stack(
    [(img.reshape(28, 28) / 255.0) for img, _ in images_and_labels_all],
    axis=0)

img0_training = img0_all[:num_training]
img0_ourtest = img0_all[num_training:]
# We could imagine doing per-pixel-feature input normalization, but this
# would have some problems due to rarely activated pixels. Let us instead
# simply make the mean pixel activation zero and standard deviation 1.
img_offset = img0_training.mean()
img_stddev = numpy.std(img0_training)
img_training = (img0_training - img_offset) / img_stddev
```

```
print(img_training.shape)
pyplot.imshow(img_training[0], cmap='gray'); pyplot.show()

# We already shuffled our images. So, we can define batches by merely reshaping
# the arrays.
batch_size = 64
num_batches = num_training // batch_size
num_training_used = num_batches * batch_size
batched_labels_1hot_training = (
    labels_1hot_training[:num_training_used].reshape(
        num_batches, batch_size, 10))

# Our classifier is built in such a way that it reshapes every single
# example's features to a vector anyhow, so we may as well feed them
# in as 28x28.
batched_img_training = img_training[:num_training_used].reshape(
    num_batches, batch_size, 28, 28)

# Let us actually build a first blunt classifier.
rng1 = numpy.random.RandomState(seed=1)
model_params = [
          # First, we (nonlinearly) map pixels to a 25-vector.
          rng1.normal(size=(28*28+1, 25), scale=1/28),
          # Then, this 25-vector to another 25-vector.
          rng1.normal(size=(25+1, 25), scale=1/5),
          # Then, once more.
          rng1.normal(size=(25+1, 25), scale=1/5),
          # Then, to a 10-vector for softmax.
          rng1.normal(size=(25+1, 10), scale=1/5)]

classifier_func = get_nn_classifier_func(model_params)


def train_one_epoch(rng,
                    params,
                    classifier_func,
                    batched_labels,
                    batched_imgs,
                    learning_rate,
                    print_every=200):
  num_batches = batched_labels.shape[0]
  batch_sequence = list(range(num_batches))
  rng.shuffle(batch_sequence)
  for num_batch, batch_index in enumerate(batch_sequence):
    loss, pred, grad = classifier_func(batched_imgs[num_batch],
                                       batched_labels[num_batch],
```

```
                                        want_gradient=True)
    if num_batch % print_every == 0:
      print(f'Batch {num_batch:4d} / {num_batches:4d}: Loss={loss:10.3f}')
    for p, delta_p in zip(params, grad):
      p -= learning_rate * delta_p

for num_epoch in range(50):
  print('### Epoch', num_epoch)
  train_one_epoch(rng1,
                  model_params,
                  classifier_func,
                  batched_labels_1hot_training,
                  batched_img_training,
                  1e-4)
```

[ ]:
```
# For a better classifier: Let's train for a bit longer.
# (Feel free to skip this.)

for num_epoch in range(50, 500):
  print('### Epoch', num_epoch)
  train_one_epoch(rng1,
                  model_params,
                  classifier_func,
                  batched_labels_1hot_training,
                  batched_img_training,
                  1e-4)
```

[ ]:
```
# Let us try out our trained classifier on a few examples from our test set.
loss1, predictions1, _ = classifier_func(
    # Need to use the same normalization function.
    (img0_ourtest - img_offset) / img_stddev,
    labels_1hot_ourtest)

pprint.pprint(predictions1[:10].round(2).tolist())
pprint.pprint(labels_1hot_ourtest[:10].tolist())
print(list(
    zip(numpy.argmax(predictions1[:20], axis=-1),
        numpy.argmax(labels_1hot_ourtest[:20], axis=-1))))

# After 500 training epochs, we get an accuracy of ~94.37%.
# Not too great, but somewhat better than what a linear model
# can hope to achieve here. After 1000 epochs, we see an
# accuracy of ~95.10% - the model is still improving with more training.
print('Accuracy:',
      numpy.mean([x == y for x, y in zip(
          numpy.argmax(predictions1, axis=-1),
          numpy.argmax(labels_1hot_ourtest, axis=-1))]))
```

```
[[0.97, 0.0, 0.0, 0.0, 0.0, 0.02, 0.0, 0.0, 0.0, 0.0],
 [0.0, 0.0, 0.01, 0.0, 0.0, 0.0, 0.0, 0.0, 0.98, 0.0],
 [0.0, 0.0, 0.0, 0.0, 0.01, 0.0, 0.0, 0.01, 0.0, 0.98],
 [0.0, 0.0, 0.0, 0.0, 0.01, 0.0, 0.0, 0.0, 0.0, 0.98],
 [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.99],
 [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.03, 0.0, 0.96],
 [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.97, 0.0, 0.03],
 [0.15, 0.0, 0.05, 0.55, 0.04, 0.01, 0.02, 0.01, 0.16, 0.02],
 [0.0, 0.99, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0],
 [0.0, 0.0, 0.0, 0.0, 0.01, 0.0, 0.99, 0.0, 0.0, 0.0]]
[[1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0],
 [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0],
 [0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0],
 [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0],
 [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0],
 [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0],
 [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0],
 [1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0],
 [0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0],
 [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0]]
[(0, 0), (8, 8), (9, 3), (9, 9), (9, 9), (9, 9), (7, 7), (3, 0), (1, 1), (6, 6),
(6, 6), (3, 3), (4, 4), (0, 0), (4, 4), (1, 1), (6, 6), (0, 0), (9, 9), (6, 6)]
Accuracy: 0.94375
```

15.2 Summary

- We have a basic trainable 'deep NN classifier' for handwritten digits.
- This setup has many of the important generic elements of a fully-connected DNN approach. Some performance-improving tricks and tweaks are still missing, though. (We can explore quite a few of these via TensorFlow Playground.)
- Here, we trained a model with more than 20k parameters to do something useful, using stochastic gradient descent.

[ ]:
```
print('Total trainable model parameters:', sum(m.size for m in model_params))
```

```
Total trainable model parameters: 21185
```

15.3 A Closer Look

Let us explore one specific detail about our current set-up. Suppose we build a MNIST-classifier much like this, where we use even more hidden layers and all make them same-with, also sticking with our atan-nonlinearity. What do the gradients then actually look like?

[ ]:
```
def explore_gradients(num_layers=7,
                      layer_width=30,
                      num_training_epochs=0,
                      learning_rate=1e-4,
                      std_weights_layer0=0.03,
                      std_biases_layer0=0.0,
```

```
                    std_weights=0.1,
                    std_biases=0.1,
                    std_weights_layer_last=0.3,
                    std_biases_layer_last=0.0,
                    num_batches=20,
                    rng=None):
  if rng is None:
    rng = numpy.random.RandomState()
  # The very first layer is a bit special - here, we go from 28x28 down
  # to layer_width.
  model_params = [
    numpy.concatenate([rng.normal(size=[28**2, layer_width],
                                  scale=std_weights_layer0),
                       rng.normal(size=[1, layer_width],
                                  scale=std_biases_layer0)],
                      axis=0)]
  for n in range(num_layers):
    model_params.append(
        numpy.concatenate([rng.normal(size=[layer_width, layer_width],
                                      scale=std_weights),
                           rng.normal(size=[1, layer_width],
                                      scale=std_biases)],
                          axis=0))
  # The very last layer also is special. We go down to width-10.
  model_params.append(
         numpy.concatenate([rng.normal(size=[layer_width, 10],
                                       scale=std_weights),
                            rng.normal(size=[1, 10],
                                       scale=std_biases)],
                           axis=0))
  classifier_func = get_nn_classifier_func(model_params)
  for num_epoch in range(num_training_epochs):
    print('# Epoch', num_epoch)
    train_one_epoch(rng,
                    model_params,
                    classifier_func,
                    batched_labels_1hot_training,
                    batched_img_training,
                    learning_rate)
  # Let us collect data for the first 10 training set batches.
  some_loss_probs_grad = [classifier_func(batched_img_training[num_batch],
                                          ␣
  ↪batched_labels_1hot_training[num_batch],
                                           want_gradient=True)
                          for num_batch in range(num_batches)]
  # We are interested in the per-layer length-of-the-gradient.
  def gradient_lengths(loss_probs_grad):
```

```
    loss, probs, grad_components = loss_probs_grad
    return [numpy.linalg.norm(g) for g in grad_components]
  per_batch_gradient_lengths = [
    gradient_lengths(l_p_g) for l_p_g in some_loss_probs_grad]
  # Let us also return the classifier.
  return numpy.array(per_batch_gradient_lengths), classifier_func

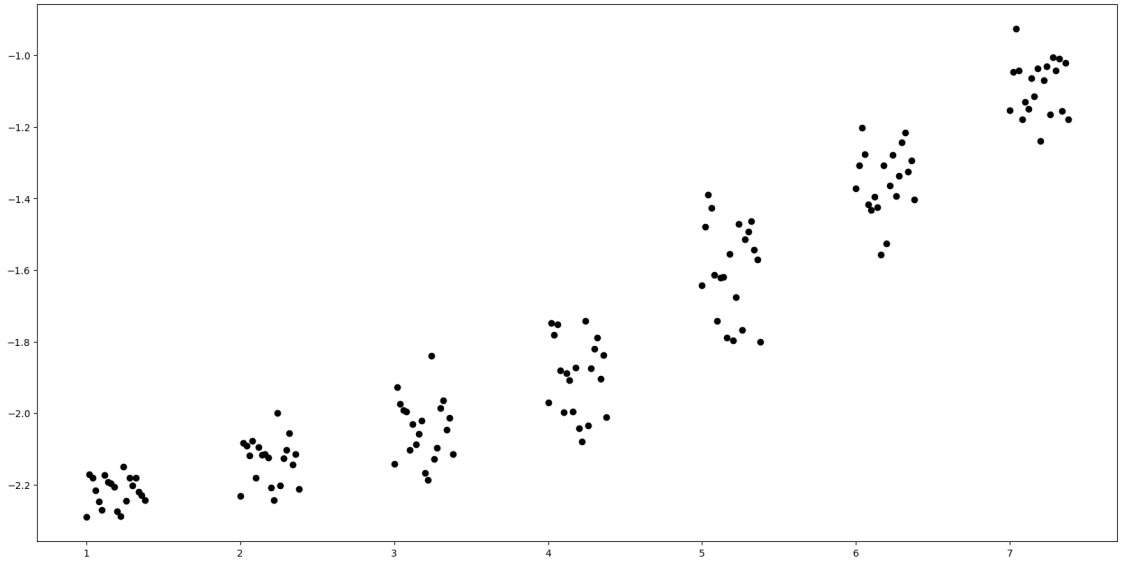
grads1, _ = explore_gradients(rng=numpy.random.RandomState(1))
grads2, _ = explore_gradients(rng=numpy.random.RandomState(1), std_weights=0.09)
```

[ ]:
```
print(grads1[:5, :5].round(5))
print('###')
print(grads2[:5, :5].round(5))
```

```
[[0.01798 0.00512 0.00585 0.00722 0.01071]
 [0.02071 0.00673 0.00826 0.01179 0.01788]
 [0.02092 0.00658 0.00808 0.01062 0.01652]
 [0.0192  0.00607 0.0076  0.01018 0.01771]
 [0.0174  0.00566 0.00836 0.01009 0.01314]]
###
[[0.00794 0.00252 0.00294 0.00398 0.00681]
 [0.00909 0.00328 0.00416 0.00656 0.01123]
 [0.0092  0.00321 0.00408 0.00587 0.01035]
 [0.00843 0.00296 0.00383 0.0056  0.01111]
 [0.00771 0.00281 0.00423 0.00541 0.00802]]
```

[ ]:
```
# We are interested in the lengths of the in-between gradients, so,
# ignoring the first and final layer.
# Let us plot these (logarithmically).
# x-coordinate = {number of weights-layer} + {number of example} * 0.02,
# y_coordinate = log(gradient_length)

pyplot.plot(
    [num_layer + num_example * 0.02
     for num_layer in range(1, grads1.shape[1] - 1)
     for num_example in range(grads1.shape[0])],
    [numpy.log(grads1[num_example, num_layer]) / numpy.log(10)
     for num_layer in range(1, grads1.shape[1] - 1)
     for num_example in range(grads1.shape[0])],
     'ok')
pyplot.show()
```



[ ]:
```
# This plot shows that, "as we backprop towards earlier layers",
# "the length of the gradient decays", so
# "late layers move a lot in training, but earlier layers only a little".
#
# Let us wrap this up and study this as a function of the weight-
# and bias-noise scaling.

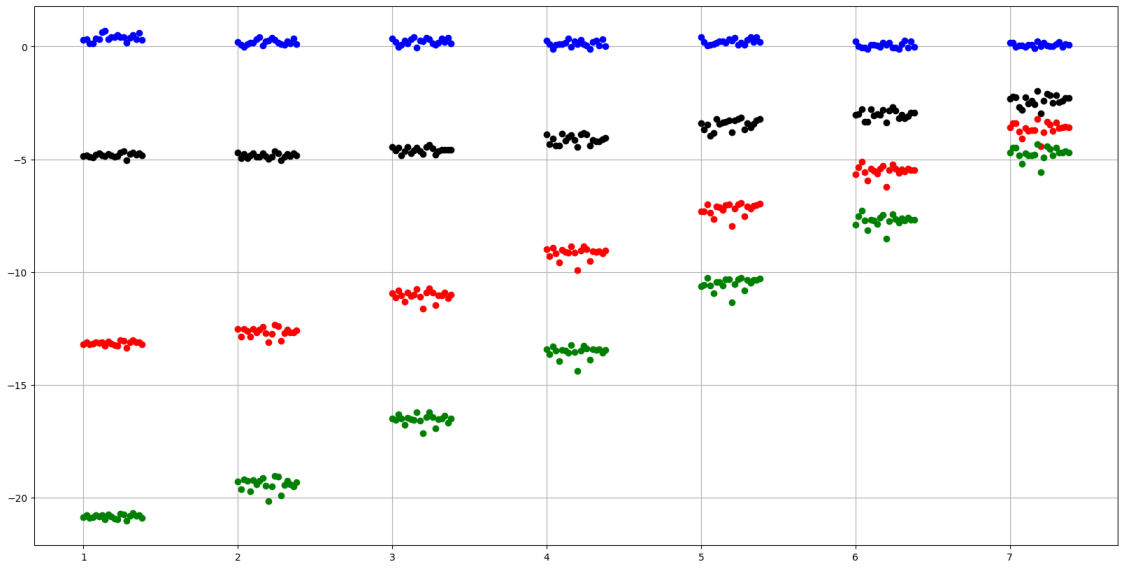
def plot_wb(seq_tag_std_weights_std_biases, num_training_epochs=0, rng_seed=0):
  for plot_tag, std_weights, std_biases in seq_tag_std_weights_std_biases:
    grads, _ = explore_gradients(rng=numpy.random.RandomState(rng_seed),
                                 std_weights=std_weights,
                                 std_biases=std_biases,
                                 num_training_epochs=num_training_epochs)
    #
    pyplot.plot(
      [num_layer + num_example * 0.02
       for num_layer in range(1, grads.shape[1] - 1)
       for num_example in range(grads.shape[0])],
      [numpy.log(grads[num_example, num_layer])
       for num_layer in range(1, grads.shape[1] - 1)
       for num_example in range(grads.shape[0])],
       plot_tag)
  pyplot.grid()
  pyplot.show()

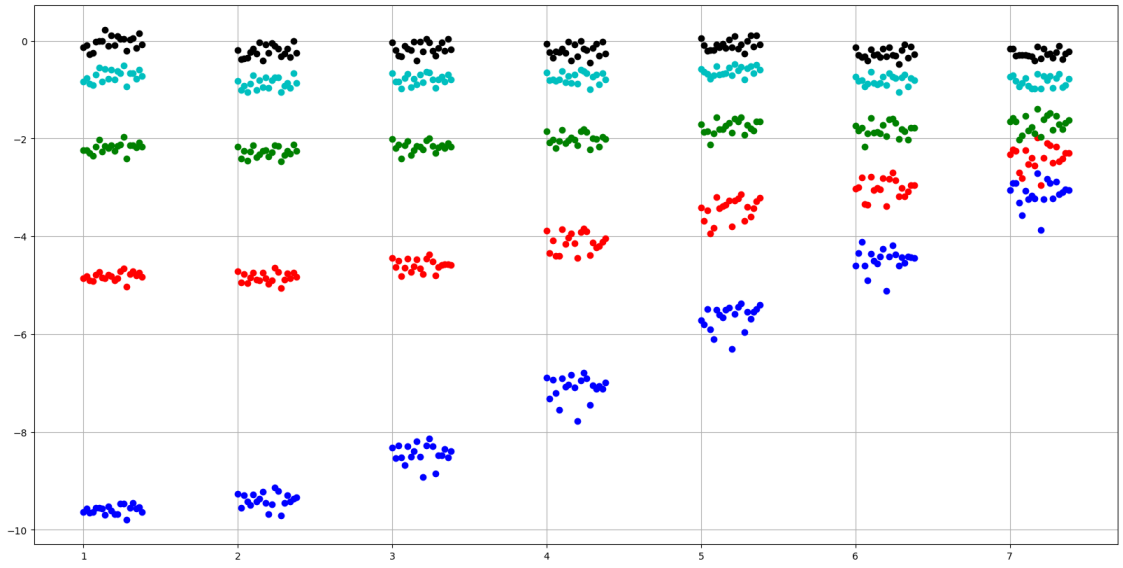

plot_wb([('ok', 0.1, 0.1),
         ('ob', 0.3, 0.1),
```

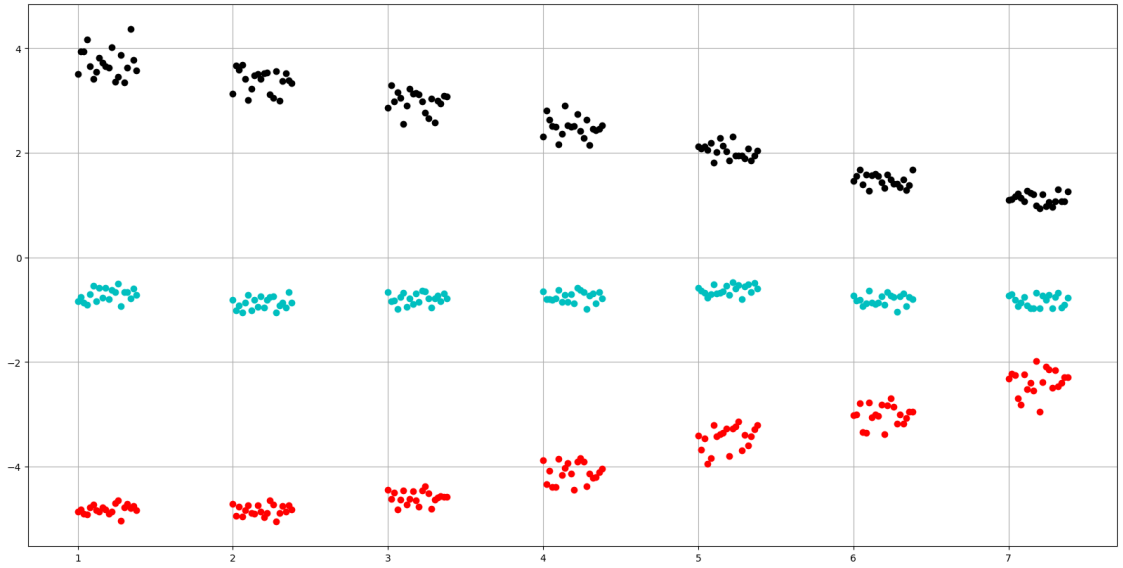
```
         ('or', 0.03, 0.1),
         ('og', 0.01, 0.1),
         ])

plot_wb([('ob', 0.05, 0.1),
         ('or', 0.1, 0.1),
         ('og', 0.15, 0.1),
         ('oc', 0.20, 0.1),
         ('ok', 0.25, 0.1),
         ])

plot_wb([('or', 0.1, 0.1),
         ('oc', 0.20, 0.1),
         ('ok', 1.0, 0.1),
         ])
```







[ ]:
```
# Let us also see what happens after a few epochs.

plot_wb([('or', 0.1, 0.1),
         ('oc', 0.20, 0.1),
         ('ok', 1.0, 0.1),
         ], num_training_epochs=5)
```

How to interpret this last plot?

For a large amount if initial-weight noise (black), the different batches (all of size 64) we use to

estimate the model-gradient produce gradients whose lenghts, for the first hidden layer, are "all over the place" (lengths from about from exp(-0.8) to about exp(2.7) or so). We introduced batches to estimate a "roughly OK" low-effort useful-for-training approximation to the model's gradient on the entire training set.

Seeing that gradient-lengths differ by a factor of 30 or so across batches makes the idea that these per-batch gradients would do anything useful look doubtful. It is plausible that we are merely adding noise at this layer in every training step.

Correspondingly, if initial-weight noise is initially very small, we get small activations which take is into the mostly-linear region of our $\tan^{-1}$ nonlinearity. Every such small-initial-weights layer then "also scales down overall activation by some factor", and it is no surprise that gradient-lengths decay as we backpropagate further away from the result and towards earlier layers.

So, it would be a mistake to assume that "we can start this with a tiny amount of noise, merely to break the symmetry", and then hope for backpropagation to adjust weights to get us something useful even with many layers - remote-from-the-loss layers will move only very little due to this gradient-decay.

Noting that in a fully-connected network with many nodes per layer, "every node is a neighbor of every node from the next layer", i.e. we have a degree of neighbor-connectivity that is higher than what we could get from neighborhood in a 4d embedding, we may wonder whether this situation would be accessible to Mean Field Theory. This has been done in https://arxiv.org/abs/1611.01232 - "Deep Information Propagation" [14] – a (in my view) seminal but wildly under-appreciated paper which likely has not received as much attention as it deserves since most members of the ML community unfortunately do not have a strong background in statistical mechanics. (It should however be noted that there would be more to say about the infinite-width limit of neural networks than what is discussed in that paper.)

A key insight here is that in order to successfully train a deep architecture, we have to pay close attention to how random weight / bias initialization is done.

If all this is actually true, then we should be able to train this "mini monster" architecture we have here into a good-quality classifier by using an informed choice of the inner-layer weight standard deviation. If we do this, we still have not really tuned initialization of random parameters for the very first and last layer, but perhaps what we have is good enough. . . ?

We still might have to do some exploration around "finding a good learning rate", though. (I did a tiny bit of manual tweaking to find a good choice. A useful recipe is to do half-order-of-magnitude adjustments to find the sweet spot between "loss barely moves at all" and "loss often increases rather than decreasing".)

[ ]:
```
grads_x, classifier_func_x = explore_gradients(
    rng=numpy.random.RandomState(0),
    std_weights=0.2,
    std_biases=0.1,
    num_training_epochs=200,
    learning_rate=3e-4)
```

```
[ ]: loss_x, predictions_x, _ = classifier_func_x(
         # Need to use the same normalization function.
         (img0_ourtest - img_offset) / img_stddev,
         labels_1hot_ourtest)

     print('Accuracy:',
           numpy.mean([x == y for x, y in zip(
               numpy.argmax(predictions_x, axis=-1),
               numpy.argmax(labels_1hot_ourtest, axis=-1))]))
     # Prints:
     # Accuracy: 0.95725
```

```
Accuracy: 0.95725
```

So, with just a few minutes of training time on a CPU, we got somewhat reasonable performance (accuracy 95.7%) out of a 8-stacked-nonlinearities (not counting softmax at the end) "deep" neural network by showing to it a collection of 48000 training examples 200 times over.

And we did it all without using any "ML framework". We merely used NumPy in a way we might just as well have used Matlab or Octave instead. If we wanted to, we could translate everything we saw here straightaway to C code. None of the computational steps involved here use any sophisticated processing or algorithms.

Naturally, this is of course still quite far from state-of-the-art. Even pre-DL revolution, error rates well below 1% were achievable for this specific task. Nevertheless, quite a few of the main ideas that allow us to perform even better can be discussed in simpler settings where we do not even have to look at code.

Specifically, these notions are readily explorable via the https://playground.tensorflow.org/ web page:

- ReLU activation function.
- L1 and L2 regularization.
- Tweaking "learning rate".

In our toy classifier, we used $\tan^{-1}$ as our nonlinearity. There are quite a few other choices which give about-similar performance. Does the activation function matter? "Yes and No" - in the sense that problematic properties of some particular choice of activation function can often be fixed by other "compensating" design decisions.

Historically, the logistic ("sigmoid") function was popolar in the early neural-networks literature. Here, we have been using a choice that "behaves symmetrically around zero".

It came to quite a surprise that, relative to these choices, the ReLU function ("rectified linear unit"), with its very simple (and scale invariant!) behavior often performs quite a bit better. This is the function $x \mapsto (x + |x|)/2$. There have been various tweaks to this, such as leaky-relu, and another refinement to this idea was ELU (replacing the negative side with an exponential).

A particularly interesting choice has been presented in https://arxiv.org/abs/1706.02515v5 - Self-normalizing neural networks [11]: This choice tries to "by design" (under reasonable assumptions) drive activations towards a "zero mean, unit variance" fixed point. However, there is a lot of

literature of the "This paper demonstrates that my nonlinearity performs better than yours" type. Pragmatically, ReLU still is a good default choice for a first shot at getting useful results.

15.4 Addendum: Thoughts on universal approximation theorems

(This was not covered on the IMPRS course given at the Albert Einstein Institute in 2022, and expresses some views of the present author that may not have the status of being consensus in the ML community.)

With neural networks, two obvious questions are: "Is some particular aspect of interest 'learnable' from the set of examples we have available" (it may well be that we would need many more examples to be sufficiently dense in space to make the aspect we are interested in approximable by our model), and "does our chosen architecture perhaps not permit learning some functions"? Clearly, a linear model would fail to learn the behavior of an exclusive-or logic gate. In general, we have "Universal Approximation Theorems" that basically state that for common popular Deep Neural Network architectures, they can learn to approximate every "reasonable" function, given sufficient capacity. (Clearly, we would however not expect any finite-size model with a simple architecture to be able to train on examples for `label = sin(1/feature)` in such a way that it would produce a main characteristic of this relation - infinitely many zeroes.) With ReLU units, this is quite intuitive - as a function of the activations that get summed and then put into a ReLU unit, this unit is linear on one side of a hyperplane and zero on the other. This means two things: First, for any given point not on any hyperplane where a ReLU starts to activate (which are finitely many, so this is a set of measure zero), the output is linear in the inputs. In a sense, a 100-layers deep ReLU network, when observed locally, "collapses to a 1-layer linear model". Second, by summing such ReLU functions - and taking ReLU outputs as inputs of other ReLU units, we effectively partition up feature-space in a mosaic of polyhedral cells (some of which will extend to infinity). Clearly, given sufficient capacity, every function that we can think of as being in some sense approximable by a continuous function that is piecewise-affine on mosaic cells is then "learnable" by a ReLU network with sufficient capacity.

This however does not mean that it would not make sense to look deeper into the general structure of data-processing inside neural networks. One interesting aspect is that we are only discussing one-dimensional nonlinearities, so there is a hidden claim here that every higher-dimensional non-linear function can be approximated by only one-dimensional nonlinearities and addition as the only binary operation (i.e. operation which "combines" data). This is true, and commonly known as the Kolmogorov-Arnold Representation Theorem (which in itself has an interesting back story and relation to Hilbert's 13th problem). There is however a problem which is especially relevant in physics-related applications where problems often have some interesting symmetry (such as: rotational symmetry).

The issue is best illustrated by example: Suppose we wanted to learn computing the determinant of a real 5×5-matrix. This is a conceptually rather simple mapping from 25 real input parameters (matrix elements) to a 1-dimensional output (the determinant) - clearly, we can write this as a 5th-degree homogeneous polynomial with $5! = 120$ summands. How would this be learned? Ultimately, this would involve learning to multiply two numbers (in many places), but since we only ever combine different internal activations by adding them, multiplication of two positive numbers would have to be synthesized from a learned approximation to a logarithm, learned independently for both inputs, then summing, followed by a learned approximation to an exponential. For signed quantities, this then becomes even more complicated. Intuitively, it is clear that "learning to com-

pute a determinant" will be possible in principle, but one would also expect that having to learn all these - from the perspective of a neural network "unnatural" - multiplication operations can easily make the size of the training set that would be required to properly learn computing a determinant (and hence, also training time) prohibitively large. The determinant is of course only one example of a polynomial geometric invariant, and the story would be similar for many a problem where we would expect learning to be able to "discover" that some derived quantity such as - say - Mandelstam variables might be useful to predict an outcome. One possible approach would be to design multiplication into the architecture, as is done with Sigma-Pi networks. Overall, these are one of many "fringe architectures" that can be employed to address very specific problems. A perhaps even more fringe (but, in the present author's view, useful) idea is to take this a step further and try to use an architecture which like a Sigma-Pi network already has an inherent concept of multiplication, but avoids much of the need for architecture-search - simply by going from a one-dimensional nonlinearity to using matrix exponentiation as a nonlinearity - our paper https://arxiv.org/abs/2008.03936 - Intelligent Matrix Exponentiation [3] characterizes properties of this nonlinearity (and in particular shows that this is indeed able to learn determinants, unlike conventional architectures - as participants are invited to verify by trying), but deliberately not going as deep into Lie group theory as one could do here, given the intended ML audience.

16 Machine Learning with TensorFlow

- In the previous unit, we have seen how to build a DNN classifier "from scratch".
- Knowing how to do this is knowledge at the level of "knowing how to derive the Euler-Lagrange equations":

  We may well find ourselves in some unusual situation where we have to "go back to first principles" to reason something out.

- Much of current ML work (model-defining, model-training) is framework-based, but there is both a blessing and a curse here:
  - Frameworks cover many common cases.
  - They also may limit our perspective to the framework's narrow field-of-view.
  - Interesting work remains to be done that will require extending frameworks, or even "working outside existing frameworks since we have to break some rules".
- Here, we will mostly focus on Google's TensorFlow library. Later, we will also look a bit into JAX.
  - TensorFlow is a well-supported evolving (but somewhat under-documented) library that is popular for ML, and has been designed for ML, but can be used for many other tasks as well. "A frigate".
  - JAX is a smaller and more experimental project that "tries to take the good bits and pieces from TensorFlow", such as "fast gradients" and "accelerated linear algebra". "A skiff".

16.1 TensorFlow for the ML Practitioner

Suppose we wanted to redo what we did by hand using the infrastructure provided by TensorFlow. Here is an "easy sailing" version.

First, let us train a model (we will come back to discussing all the nice things that we are getting for free here thanks to TensorFlow).

[ ]:
```
# We will need this PyPI module later
!pip install tf2onnx


import os
import time

import numpy
import tensorflow as tf
import tensorflow_datasets as tfds


# Loading the training and test set. We split the training-set into 'training',
# 'validation', and 'extra' for ad-hoc purposes.
def get_training_datasets():
  (ds_train_raw,
   ds_validation_raw,
   ds_extra_raw,
   ds_test_raw), ds_info = tfds.load(
      'mnist',
      split=['train[:75%]', 'train[75%:99%]', 'train[99%:]', 'test'],
      shuffle_files=True,
      as_supervised=True,
      with_info=True)
  total_num_examples = sum(s.num_examples for s in ds_info.splits.values())
  #
  def normalize_image(image, label):
    """Normalizes images."""
    return tf.cast(image, tf.float32) / 255., label
  #
  def transform(ds):
    return (ds
            .map(normalize_image, num_parallel_calls=tf.data.AUTOTUNE)
            .cache()
            # Shuffle buffer size needs to be at least as large as the number
            # of examples - but does not matter much otherwise.
            .shuffle(total_num_examples, seed=0)
            .batch(32)
            .prefetch(tf.data.AUTOTUNE))
  #
  return (transform(ds_train_raw),
          transform(ds_validation_raw),
          transform(ds_extra_raw),
          transform(ds_test_raw))
```

```
ds_train, ds_validation, ds_extra, ds_test = get_training_datasets()
```

[ ]:
```
# Training and test sets are iterators.
# For later, we save part of the `ds_extra` dataset to the filesystem.

# The os.access() check makes this cell idempotent.
if os.access('training_examples.npz', os.R_OK):
  print('Small sample dataset already was saved to filesystem.')
else:
  sample_batches = list(ds_extra.take(10).as_numpy_iterator())
  sample_batches_images = numpy.stack(
    [image_data for image_data, labels in sample_batches], axis=0)
  sample_batches_labels = numpy.stack(
    [labels for image_data, labels in sample_batches], axis=0)
  #
  # `sample_batches_images` is a `[num_batches, 32, 28, 28, 1]`-array:
  # `num_batches` batches of 32 images each which are 28x28 with one
  #  color-channel.
  # `training_batches_labels` is a `[num_batches, 32]`-array:
  #  One label per batch per image in the batch.
  print('Shapes:', sample_batches_images.shape, sample_batches_labels.shape)
  print('Labels:\n', sample_batches_labels, sep='')
  #
  numpy.savez_compressed('training_examples.npz',
                         images=sample_batches_images,
                         labels=sample_batches_labels)
```

```
Shapes: (10, 32, 28, 28, 1) (10, 32)
Labels:
[[6 2 3 1 6 9 3 6 1 2 5 4 2 1 5 8 5 7 6 9 3 8 6 4 1 5 9 8 3 2 6 9]
 [7 2 9 9 9 5 4 4 0 8 2 8 7 4 1 2 6 8 3 4 5 8 0 5 2 8 9 7 5 8 5 7]
 [3 9 6 9 1 3 2 3 0 7 7 2 4 6 1 7 4 3 3 9 4 1 1 2 1 4 6 2 2 8 6 4]
 [2 8 4 3 3 5 6 4 1 1 1 5 7 7 5 7 4 5 1 5 7 3 0 1 1 2 8 8 5 1 7 1]
 [0 1 3 3 5 6 0 3 9 1 7 0 7 3 1 9 4 5 5 8 8 6 1 7 3 7 2 6 7 1 7 3]
 [7 9 0 8 8 7 4 3 6 5 8 8 9 8 1 7 3 4 1 9 5 7 8 1 9 4 0 7 2 3 4 5]
 [2 0 7 2 8 6 2 3 6 1 9 2 7 4 4 8 4 1 5 2 7 2 0 8 8 3 9 3 3 0 3 4]
 [2 3 8 5 0 1 6 5 5 0 8 5 9 9 4 8 8 1 4 4 9 8 4 4 6 5 3 0 8 8 1 7]
 [6 0 1 6 8 2 3 9 4 8 1 1 0 4 4 2 3 4 1 0 5 1 9 0 0 6 2 7 5 2 7 9]
 [1 8 4 9 2 1 9 1 0 8 0 1 4 4 7 5 3 8 2 9 9 4 1 6 2 5 8 1 1 3 7 3]]
```

[ ]:
```
### Training a model.

model = tf.keras.models.Sequential([
  tf.keras.layers.Flatten(input_shape=(28, 28)),
  tf.keras.layers.Dense(50, activation='tanh'),
  tf.keras.layers.Dense(50, activation='tanh'),
```

```
  tf.keras.layers.Dense(50, activation='tanh'),
  tf.keras.layers.Dense(50, activation='tanh'),
  tf.keras.layers.Dense(10)
])

# Participant Exercise: Try out other architectures, such as...:
#
# model = tf.keras.models.Sequential([
#   tf.keras.layers.Flatten(input_shape=(28, 28)),
#   tf.keras.layers.Dense(80, activation='relu'),
#   tf.keras.layers.Dense(80, activation='relu'),
#   tf.keras.layers.Dense(80, activation='relu'),
#   tf.keras.layers.Dense(10)
# ])
#
# How simple can we make this and still get >95% accuracy?


model.compile(
    optimizer=tf.keras.optimizers.Adam(1e-4),
    loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
    metrics=[tf.keras.metrics.SparseCategoricalAccuracy()],
)

model.fit(
    ds_train,
    epochs=10,
    validation_data=ds_validation)
```

```
Epoch 1/10
1407/1407 [==============================] - 33s 10ms/step - loss: 0.7655 -
sparse_categorical_accuracy: 0.8082 - val_loss: 0.4042 -
val_sparse_categorical_accuracy: 0.8922
Epoch 2/10
1407/1407 [==============================] - 6s 4ms/step - loss: 0.3325 -
sparse_categorical_accuracy: 0.9089 - val_loss: 0.3063 -
val_sparse_categorical_accuracy: 0.9135
Epoch 3/10
1407/1407 [==============================] - 6s 4ms/step - loss: 0.2651 -
sparse_categorical_accuracy: 0.9238 - val_loss: 0.2643 -
val_sparse_categorical_accuracy: 0.9244
Epoch 4/10
1407/1407 [==============================] - 6s 4ms/step - loss: 0.2270 -
sparse_categorical_accuracy: 0.9344 - val_loss: 0.2390 -
val_sparse_categorical_accuracy: 0.9310
Epoch 5/10
1407/1407 [==============================] - 6s 4ms/step - loss: 0.1996 -
sparse_categorical_accuracy: 0.9425 - val_loss: 0.2149 -
```

```
val_sparse_categorical_accuracy: 0.9373
Epoch 6/10
1407/1407 [==============================] - 7s 5ms/step - loss: 0.1778 -
sparse_categorical_accuracy: 0.9479 - val_loss: 0.1988 -
val_sparse_categorical_accuracy: 0.9422
Epoch 7/10
1407/1407 [==============================] - 6s 4ms/step - loss: 0.1597 -
sparse_categorical_accuracy: 0.9541 - val_loss: 0.1862 -
val_sparse_categorical_accuracy: 0.9474
Epoch 8/10
1407/1407 [==============================] - 6s 4ms/step - loss: 0.1447 -
sparse_categorical_accuracy: 0.9580 - val_loss: 0.1788 -
val_sparse_categorical_accuracy: 0.9469
Epoch 9/10
1407/1407 [==============================] - 7s 5ms/step - loss: 0.1326 -
sparse_categorical_accuracy: 0.9613 - val_loss: 0.1684 -
val_sparse_categorical_accuracy: 0.9513
Epoch 10/10
1407/1407 [==============================] - 6s 4ms/step - loss: 0.1218 -
sparse_categorical_accuracy: 0.9649 - val_loss: 0.1631 -
val_sparse_categorical_accuracy: 0.9525
```

[ ]:
```
<keras.callbacks.History at 0x7dd008367700>
```

We now have a trained model. Perhaps not one with quite state-of-the-art performance, but at least a model that clearly seems to know something about the task it was being built for.

Let us see how to: * Get information about the trained model. * Save it to a file and load it again. * Actually use it to make predictions.

[ ]:
```
model.summary()

# Saving to a `*.h5` file will make tf-Keras use the HDF5 format
# for the saved model.
model.save("mnist_model.h5")

print('######')
!ls -lah mnist_model*

reloaded_model = tf.keras.models.load_model("mnist_model.h5")


# Let us use one batch of examples we extracted earlier:
examples_images, examples_labels = (
    sample_batches_images[0, ...], sample_batches_labels[0, ...])

def predict(image, verbose=False):
  if image.size != 28*28:
```

```
    raise ValueError('Expecting input data to provide 28x28 pixels.')
  logits = model.predict(image.reshape(1, 28, 28), verbose=verbose)
  if verbose:
    print('Logits:', logits.round(3))
  return numpy.argmax(logits)

# We could run predictions on an entire batch, but here process
# individual images.
for num_example, (image, label) in enumerate(zip(examples_images,
                                                 examples_labels)):
  predicted = predict(image, verbose=False)  # Feel free to set verbose=True
  ok = 'OK' if predicted == label else 'BAD'
  print(f'Example Image {num_example:2d}: '
        f'predicted={predicted}, actual={label}  - {ok}')
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 flatten (Flatten)           (None, 784)               0

 dense (Dense)               (None, 50)                39250

 dense_1 (Dense)             (None, 50)                2550

 dense_2 (Dense)             (None, 50)                2550

 dense_3 (Dense)             (None, 50)                2550

 dense_4 (Dense)             (None, 10)                510

=================================================================
Total params: 47,410
Trainable params: 47,410
Non-trainable params: 0
_________________________________________________________________
######
-rw-r--r-- 1 root root 605K Aug 22 11:34 mnist_model.h5
Example Image  0: predicted=6, actual=6  - OK
Example Image  1: predicted=2, actual=2  - OK
Example Image  2: predicted=3, actual=3  - OK
Example Image  3: predicted=1, actual=1  - OK
Example Image  4: predicted=6, actual=6  - OK
Example Image  5: predicted=9, actual=9  - OK
Example Image  6: predicted=3, actual=3  - OK
Example Image  7: predicted=6, actual=6  - OK
Example Image  8: predicted=1, actual=1  - OK
Example Image  9: predicted=7, actual=2  - BAD
```

```
Example Image 10: predicted=5, actual=5  - OK
Example Image 11: predicted=4, actual=4  - OK
Example Image 12: predicted=2, actual=2  - OK
Example Image 13: predicted=1, actual=1  - OK
Example Image 14: predicted=5, actual=5  - OK
Example Image 15: predicted=8, actual=8  - OK
Example Image 16: predicted=5, actual=5  - OK
Example Image 17: predicted=7, actual=7  - OK
Example Image 18: predicted=6, actual=6  - OK
Example Image 19: predicted=7, actual=9  - BAD
Example Image 20: predicted=3, actual=3  - OK
Example Image 21: predicted=8, actual=8  - OK
Example Image 22: predicted=6, actual=6  - OK
Example Image 23: predicted=4, actual=4  - OK
Example Image 24: predicted=1, actual=1  - OK
Example Image 25: predicted=5, actual=5  - OK
Example Image 26: predicted=9, actual=9  - OK
Example Image 27: predicted=8, actual=8  - OK
Example Image 28: predicted=3, actual=3  - OK
Example Image 29: predicted=2, actual=2  - OK
Example Image 30: predicted=6, actual=6  - OK
Example Image 31: predicted=9, actual=9  - OK
```

While we are at it: A 'trained ML model' is a bit like an 'electronics module': We would typically like to know how to use this as a component in some larger engineering design (/ product).

It is quite possible to use trained TensorFlow models on smartphones, advanced microcontrollers, or merely make them part of some compiled application. Often, a good approach is to convert the model to "TensorFlow Lite" form for deployment.

There are TFLite libraries for various systems/architectures to then load a model, feed input to it, and obtain predictions from it. These exist for: microcontrollers, tiny computers such as the Raspberry Pi, Android apps, iOS apps, compiled binaries, etc.

Here, we will just sketch how this looks like using again Python-TFLite - so, we are not quite cutting the Python umbilical cord yet. We will first save our model in TFLite form, and then switch over from using TensorFlow to using only the TFLite module.

[ ]:
```
# Converting the model to TFLite ("TensorFlow Lite") form.

model = tf.keras.models.load_model("mnist_model.h5")
converter = tf.lite.TFLiteConverter.from_keras_model(model)
tflite_model = converter.convert()

# Save the model.
with open('mnist_model.tflite', 'wb') as f:
  f.write(tflite_model)

!ls -lah 'mnist_model.tflite'
```

```
WARNING:absl:Found untraced functions such as _update_step_xla while saving
(showing 1 of 1). These functions will not be directly callable after loading.

-rw-r--r-- 1 root root 189K Aug 22 11:34 mnist_model.tflite
```

For later, we can also convert the trained model to `.onnx` format, which is understood by other projects as a representation of a computational graph. For graphs that do not use too exotic computational operations, this should generally work. At the time of this writing, we cannot convert from a model saved in HDF5 format, so we need to save our model again in TensorFlow's own format. Rather than importing the `tf2onnx` module here, we perform conversion in a subprocess, executed via a shell escape.

[ ]:
```
model.save("mnist_model_tf")

!python -m tf2onnx.convert --saved-model mnist_model_tf --output mnist_model.onnx
!ls -lah mnist_model.*
```

```
WARNING:absl:Found untraced functions such as _update_step_xla while saving
(showing 1 of 1). These functions will not be directly callable after loading.

/usr/lib/python3.10/runpy.py:126: RuntimeWarning: 'tf2onnx.convert' found in
sys.modules after import of package 'tf2onnx', but prior to execution of
'tf2onnx.convert'; this may result in unpredictable behaviour
  warn(RuntimeWarning(msg))
2023-08-22 11:42:04,202 - WARNING - '--tag' not specified for saved_model. Using
--tag serve
2023-08-22 11:42:04,486 - INFO - Signatures found in model: [serving_default].
2023-08-22 11:42:04,486 - WARNING - '--signature_def' not specified, using first
signature: serving_default
2023-08-22 11:42:04,486 - INFO - Output names: ['dense_4']
2023-08-22 11:42:04,566 - INFO - Using tensorflow=2.12.0, onnx=1.14.0,
tf2onnx=1.15.0/6d6b6c
2023-08-22 11:42:04,566 - INFO - Using opset <onnx, 15>
2023-08-22 11:42:04,572 - INFO - Computed 0 values for constant folding
2023-08-22 11:42:04,586 - INFO - Optimizing ONNX model
2023-08-22 11:42:04,629 - INFO - After optimization: Cast -1 (1->0), Identity -2
(2->0)
2023-08-22 11:42:04,631 - INFO -
2023-08-22 11:42:04,631 - INFO - Successfully converted TensorFlow model
mnist_model_tf to ONNX
2023-08-22 11:42:04,631 - INFO - Model inputs: ['flatten_input']
2023-08-22 11:42:04,631 - INFO - Model outputs: ['dense_4']
2023-08-22 11:42:04,631 - INFO - ONNX model is saved at mnist_model.onnx
-rw-r--r-- 1 root root 605K Aug 22 11:34 mnist_model.h5
-rw-r--r-- 1 root root 190K Aug 22 11:42 mnist_model.onnx
-rw-r--r-- 1 root root 189K Aug 22 11:34 mnist_model.tflite
```

Let us actually download the `.tflite` and also `.onnx` file locally...

[ ]:
```
import google.colab.files
google.colab.files.download('mnist_model.tflite')
google.colab.files.download('mnist_model.onnx')
```

[ ]:
```
# NOTE: For some versions of TensorFlow and TFLite, it is not possible
# to import `tflite` and `tensorflow` into the same Python process.
#
# This normally is not even needed, given that `tf` has a `tf.lite` sub-module,
# but here we want to demonstrate using TFlite only and not the full-bloum
# TensorFlow module.
#
# If executing this cell fails, then restarting the Colab runtime will
# replace the running Python interpreter with a new one while retaining
# on-filesystem state of the virtual machine. So, it might be necessary to
# do [Runtime] -> [Restart Runtime] (Short-cut: Control-M dot) before
# continuing with this notebook by executing this cell.
#
# Since the runtime system may have been restarted here,
# we re-import the modules that we will need going forward.

# Note: On non-colab systems, if `tflite-runtime` is not yet available
# for too-recent a CPython version, this might require e.g.
# python3.10 -m pip install tflite-runtime
!pip install tflite-runtime

import time
import numpy
import tflite_runtime.interpreter as tflite

reloaded_examples = numpy.load('training_examples.npz')
sample_batches_images =  reloaded_examples['images']
sample_batches_labels =  reloaded_examples['labels']
```

```
Requirement already satisfied: tflite-runtime in /usr/local/lib/python3.10/dist-
packages (2.13.0)
Requirement already satisfied: numpy>=1.21.2 in /usr/local/lib/python3.10/dist-
packages (from tflite-runtime) (1.23.5)
```

[ ]:
```
def get_mnist_tflite_predictor(model_path):
  interpreter = tflite.Interpreter(model_path=model_path)
  interpreter.allocate_tensors()
  input_details = interpreter.get_input_details()
  output_details = interpreter.get_output_details()
  def fn_predict(in_data, verbose=False):
    t0 = time.time()
    # Note that this is not reentrant! Different invocations of the current
    # function use the same `interpreter`, and mutate its state by
    # "setting the input". So, if multithreading executed called a function
```

```
    # that does this concurrently more-than-once-at-the-same-time, things
    # would go wrong.
    interpreter.set_tensor(input_details[0]['index'],
                           in_data.reshape(1, 28, 28))
    interpreter.invoke()
    t1 = time.time()
    output_data = interpreter.get_tensor(output_details[0]['index'])
    result = numpy.squeeze(output_data)
    if verbose:
      print(f'(t={(t1-t0)*1000:.3f} msec):', result.round(3))
    return result
  return fn_predict

mnist_tflite_predictor = get_mnist_tflite_predictor(
    model_path='mnist_model.tflite')

print('Predicted: ',
      numpy.argmax(mnist_tflite_predictor(sample_batches_images[0, 0, ...],
                                          verbose=True)),
      ' -  Actual: ', sample_batches_labels[0, 0])
```

```
(t=0.906 msec): [-4.890e-01 -1.157e+00 -2.000e-03 -3.020e+00  8.140e-01
9.350e-01
  8.952e+00 -8.248e+00 -7.890e-01 -3.224e+00]
Predicted:  6  -  Actual:  6
```

16.1.1 What TensorFlow did for us here

- For simple multi-layer architectures:
  - Provide us with an extremely simple way to specify a neural network.
  - Automatically pick reasonable defaults for the distribution of randomized initial weights and biases.
  - Hide all the subtleties around computing fast gradients.
- Offer a convenient way to specify how we want to utilize gradients for optimization.

  Earlier: 'multiply gradient with a small factor and take a small step in the opposite direction (going down)'.

  Here: pick a very simple yet quite effective alternative strategy known as 'Adam Optimization'.
- Allow us a convenient way to specify a loss function - here directly from logits.

  (Earlier, we had to go from logits to probabilities, do softmax, and then hand-backpropagate it all).
- Provide convenience functions for loading and transforming input data, which include very substantial performance optimizations.
- Handle setting up the computation in such a way that training can optionally run on CPU, GPU, or TPU(!)

- Provide us with a straightforward way to save and deploy a trained model.
- Allow us to specify performance metrics to track during training.

16.1.2 What we have not seen yet

- "The scaffolding between this high-level perspective and the low-level approach we discussed earlier."
- How to put unusual data-processing into TensorFlow so that we can still use high level infrastructure like model-saving.
- How to use the "tensor arithmetics with fast gradients" machinery inside TensorFlow to do non-ML physics.
- More tuning and tweaks we can apply to improve performance (L2 regularization, architectural elements such as: dropout layers, convolutional+max-pooling layers).

16.1.3 Additional remarks

- On 'weight initialization': TensorFlow by default uses "Glorot [4] (or 'Xavier') initialization" - basic idea: if N normal distributed inputs get summed over, use a 1/sqrt(N) scaling factor for random weight initialization.
- Here, we are using the 'test set' to obtain performance metrics. This is generally a bad idea, since it leads to applying tweaks based on information obtained by "peeking at test set performance".

  If we want to report proper ability-to-generalize, we must not use observations on the test set to make model tuning decisions.

  (The right thing to do here is to lock away the "test set" and split the for-training examples into an actual 'training set for computing gradients' and a 'validation set' for making tuning decisions.)

  Overall, this example is mostly about showing "how we can wire things up", and in this context, this is tolerable, since we are not out to "break the world record on MNIST classification".
- Nowadays, a ML system doing MNIST-classification is a bit like a TV screen showing a test image - it merely indicates that we built something which allows data to flow the way it should, and we might be able to identify some glaring problems.

16.1.4 Architecture Improvements

Before we "open the engine hood" and take a deeper look at the technology, let us discuss a few easy-to-explain ideas around "what can we do here to further improve image classifier performance?"

There are some simple general ideas that are reasonably easy-to-explain in just one or two paragraphs - if we accept that the "cheap heuristic explanations" we are giving might actually be a little bit off. At this level, this is more "cooking advice" than a thorough discussion of deeper characteristics.

L2 Regularization Now that we have seen how to train deep architectures: What would we expect to happen if we substantially over-sized or under-sized a model (in terms of number of layers and also units per layer) relative to what experimentation tells us is the point where increasing model size brings no further benefit?

Under-sizing is easy to understand: The model will not be able to extract and process all the features that are in principle available, and will have to make hard choices what to ignore.

Over-sizing is more interesting: If we massively over-sized the model, and used a fixed-size training set, it would at some point have enough capacity to just memorize every training example - and it may memorize such examples in weird ways, such as "If the top right pixel is green and the top left pixel is brighter than its neighbors, then we know we are looking at example #1735". Naturally, this then messes up the model's ability to generalize. *In particular*, if there is any classification error in the training set labels, the model will not do what an under-sized model would do - "if I cannot do well and have to take a hit, I perhaps should take one on this crazy outlier in the data". Rather, it would try to come up with a very fancy "explanation" why the outlier (and some region around it) is "really special".

If we built a model with only linear activation functions, there would be no point going to two or more layers - since "all transformations are linear". So, the power of deep architectures is closely related to their ability to put nonlinearities to good use. If we used an architecture with tanh-activation as the nonlinearity on hidden layers, and a final linear scaling layer, then tanh-s that receive small-magnitude input would behave mostly-linear.

With this in mind, it appears natural to try an approach where "we give the network the ability to use many nonlinearities, but we attach a small price to it, so the network will only decide to actually use that power if that brings a benefit".

Now, we are in an "optimize performance, but also do X" (i.e. try not to use large weights) situation. We still want to consider ML training as a numerical optimization problem - but that then means that we have to slightly deviate from the idea of "what we optimize for is performance" towards "our loss function is a weighted sum of contributions, where one measures performance, and the others are about other relevant aspects, such as minimizing some quantity that we believe to be a useful proxy for the model inclination to avoid unnecessarily complex explanations".

The basic idea behind "L2 regularization" is: Let us add a term to the total loss that is the sum-squared of weights. The model can use nonlinearity, and "smallish weights are still cheap", but making a weight large is considered costly.

The effect is nicely illustrated in these two TensorFlow Playground training runs: Two large-capacity models trained on the same "simple explanation but noisy data" dataset, having the same outliers. Without L2 regularization, we see a tendency to "draw a complicated border to explain the training set". With L2 regularization, we get a nicer border. As expected, the complex border is related to "overfitting", and we do see an accuracy gap between training and test set.

(See screenshots "L2 Regularization" on "supplementary material" document.)

ReLU activation Traditionally, the earliest nonlinearity to be used widely in neural network research was the 'sigmoid' $x \mapsto 1/(1+\exp(-x))$ (this again is of course just the logistic function). The thinking here was that this should serve as a crude model for biological neurons that still allows us to do backpropagation: Zero output for low activation, saturation for high activation, but we want a continuous function.

It so turns out that one can in principle use just about every "not too wild" nonlinearity for Deep Neural networks, even functions such as $x \mapsto x^2$ or $x \mapsto \exp(-x^2/2)$. Overall, this might not be too surprising at least for smooth such functions - "if we zoom in at some point, we will see linear

behavior, if we zoom out a bit, we will see quadratic corrections, but higher corrections will still be mostly-negligible".

It somewhat came to a surprise to discover that an extremely simple nonlinearity, which furthermore happens to be scale invariant, usually performs really well in comparison - the "rectified linear unit", $x \mapsto (x + |x|)/2$. There was a natural drive towards such a very simple function from the desire to avoid complicated numerical calculations such as doing an exponential - and it stuck when it showed good performance.

More recently, there have been variations on the topic, such as ELU and SELU, but ReLU remains an important workhorse.

It is interesting to ponder what the functions described by exclusively-ReLU deep networks look like: these are continuous functions that are affine-linear on (generalized) polyhedral cells. ("Generalized" since they may extend to infinity.) Naturally, "gradients just propagate through, even to very deep layers".

The power of a deep ReLU network is nicely illustrated in the TensorFlow Playground by doing a "swiss roll with noisy data" problem using only the x/y coordinate as input features, and going for a maximum-capacity network, even without any regularization - but one really has to watch the training process to appreciate this:

(See screenshot "ReLU Activation" on "supplementary material" document.)

Dropout Regularization Looking at some TensorFlow Playground training runs for the "swiss roll" problem that use the same quite noisy input data, but started from differently initialized random weights, it is not surprising that we get roughly-comparable classifier performance, but both models made different doubtful decisions about where to make the contour look complicated to net a few more examples and get slightly lower loss on the training set.

So, each model will have "quirks". Naturally, we would expect that if we just did what we discussed earlier, combining the assessments of different (perhaps not quite independent) "experts", such as majority-vote-of-5-differently-trained-models, we should be able to average out such "structure hallucinations". Of course, training and deploying multiple models is expensive, so: can we perhaps use some variant of this idea which retains some of the "averaging over different models" property while not being so expensive?

One idea here is to use "dropout" layers: During the training process, we keep randomly disabling some units (but correspondingly scaling up the total input to a unit that has temporarily lost some of its input units) in changing patterns. Effectively, we are training an exponentially large family of networks (since with N "faulty" units, we have $\sim 2^N$ ways to disable half of them) which are all obtained from a "master network" by turning off some units.

Another way to think about this is that "dropout" punishes complex co-adaptation of many units. If a complex hypothesis is realized by having some specific set of 10 units activate in a particular pattern, "dropout" makes it unlikely that they all are available at the same time, so, less complex hypotheses that only require two or three units to cooperate are favored over intricate explanations.

(See screenshots "Dropout" on "supplementary material" document.)

Early Stopping This is in the category of "sometimes it helps, but sometimes we observe the opposite, and it helps more to keep training even when the model stopped learning anything".

The basic idea is that it can happen that a model first learns the overall structure of the problem, those aspects that generalize well, but as it keeps being fed the training examples, it will increasingly fine-tune on accidental properties of the training set - so, we should be able to limit overfitting by stopping the training process early, typically according to some heuristics.

Small Batch Sizes This point is controversial - there are indications that, while there is some truth to this advice, the opposite also holds.

Overall, neural network training amounts to finding some minimum (in weight/bias parameter space) of a "loss: function that has many local minima (at the very least since we can always permute/relabel units and get an equivalent network). If we use smallish batch sizes (perhaps 32 or 64) to estimate gradients, that makes gradients inherently noisy, and this noisiness makes us not see narrow basins in the loss function, going for larger,"more robust" minima instead.

Convolutional Architectures So far, we fed our classifiers one-dimension-per-pixel data vectors.

Now, if we would in advance pick an image-scrambling permutation of pixels and applied this in the same way to all training and test examples, this would make the problem no more difficult for the ML architectures we have seen so far, but would make the problem *much* harder for a human. So, clearly, the problem has extra structure which we do not even remotely exploit yet.

The MNIST dataset is normalized to always have the ink-center-of-gravity in the middle of the image, and also in some other ways. This makes this item somewhat a case of "this happens to work also on MNIST, despite the major reasons why this is a good thing applying to a limited extent". Sticking with MNIST, one way to think about the problem is that we might be able to reduce "overfitting" (so, mistaking accidental structure observed in the training set for relevant) by applying some data-reduction that we would expect to reduce such accidental features. Obviously, digits are a quite anthropomorphic solution to the problem of communicating data between humans - in the sense that doing the same between computers, we would likely go for a more barcode or QR-code like solution. So, unsurprisingly, digits work reasonably well even for people with slightly bad eyesight - they do not contain relevant fine detail that must be exactly right.

So, one might wonder whether blurring the image a bit could help with classification. A "blurring" transform basically amounts to performing a convolution with some kernel, such as a Gaussian. This leaves the question: What kernel to use? If we were to answer "this is ML, so we might simply treat the kernel's parameters as learnable and use gradient descent to find out what a good kernel might be", and perhaps add "let's blur in more than one way at the same time", we basically have invented "Convolutional Neural Networks".

Another way to think about this: If we humans look at an object, it mostly does not matter much whether it sits right in the center of our field of vision, or is a tiny bit displaced to the left, right, or up, or down. So, for many image classifiaction problems, there is some inherent translational symmetry in the task: If I want to know whether there is a frog in an image, then it should not matter for detecting the frog if the camera "pointed two pixels further to the left" when the picture was taken. So, we might want to do localized feature-extraction that is only sensitive to some small windowed region in the image, and slide that window over the entire image, both horizontally and vertically. Here, "each window gets treated the same", in the sense that we use the same weights independent of window-position. So, in training, if we have frogs in 4 out of 16 batch-examples, then such a set-up will try to tweak the convolution kernel(s) in such a way that they become sensitive

to the differences between images-with-frogs and images-without-frogs, irrespective of how we have to place the window(s) to best fit the frogs.

Right after a convolutional layer, which convolves window-wise with a collection of learnable kernels, one typically puts a resolution-reducing layer, often max-pooling. About max-pooling, Geoffrey Hinton has (in)famously said: "The pooling operation used in convolutional neural networks is a big mistake and the fact that it works so well is a disaster." Here, we contend ourselves with observing that "it often works quite well."

[ ]:
```
# Before we proceed with `tf` code again, we might have to re-start the runtime
# system and re-run the very first cell of this notebook, which re-imports
# TF and other modules, and loads the dataset.
```

[ ]:
```
# Let us put some of these ideas to the test.
# Example adjusted from: https://keras.io/examples/vision/mnist_convnet/


cnn_model = tf.keras.models.Sequential(
    [
        tf.keras.layers.Input(shape=(28, 28, 1)),
        tf.keras.layers.Conv2D(32, kernel_size=(3, 3), activation="relu"),
        tf.keras.layers.MaxPooling2D(pool_size=(2, 2)),
        tf.keras.layers.Conv2D(64, kernel_size=(3, 3), activation="relu"),
        tf.keras.layers.MaxPooling2D(pool_size=(2, 2)),
        tf.keras.layers.Flatten(),
        tf.keras.layers.Dropout(0.5),
        tf.keras.layers.Dense(50, activation='relu'),
        tf.keras.layers.Dense(10),
    ]
)

cnn_model.compile(
    optimizer=tf.keras.optimizers.Adam(1e-4),
    loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
    metrics=[tf.keras.metrics.SparseCategoricalAccuracy()],
)

if True:
  # This can take about half an hour or so. Feel free to disable.
  cnn_model.fit(ds_train, epochs=30, validation_data=ds_validation)
```

[ ]:
```
if True:
  # Some extra training, for refinement - feel free to disable.
  cnn_model.fit(ds_train, epochs=2, validation_data=ds_validation)
```

```
Epoch 1/2
1407/1407 [==============================] - 58s 41ms/step - loss: 0.0293 -
sparse_categorical_accuracy: 0.9906 - val_loss: 0.0311 -
val_sparse_categorical_accuracy: 0.9904
```

```
Epoch 2/2
1407/1407 [==============================] - 57s 41ms/step - loss: 0.0286 -
sparse_categorical_accuracy: 0.9906 - val_loss: 0.0306 -
val_sparse_categorical_accuracy: 0.9906
```

[ ]:
```
cnn_model.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d_2 (Conv2D)           (None, 26, 26, 32)        320

 max_pooling2d_2 (MaxPooling  (None, 13, 13, 32)       0
 2D)

 conv2d_3 (Conv2D)           (None, 11, 11, 64)        18496

 max_pooling2d_3 (MaxPooling  (None, 5, 5, 64)         0
 2D)

 flatten_1 (Flatten)         (None, 1600)              0

 dropout_1 (Dropout)         (None, 1600)              0

 dense_2 (Dense)             (None, 50)                80050

 dense_3 (Dense)             (None, 10)                510

=================================================================
Total params: 99,376
Trainable params: 99,376
Non-trainable params: 0
_________________________________________________________________
```

What we have here may well be superhuman ability at reading handwritten digits which we can readily deploy as a 400 KB model file on even quite cheap microcontrollers. This certainly is interesting.

However, given that we still did not even try hard yet, this illustrates another point: "MNIST is in general too simple to demonstrate superiority of some ML method" and "just about every idea works well on MNIST".

[ ]:
```
# Let us download/save HDF5 and tflite forms of this model.

from google.colab import files

!rm -rf mnist_cnn_model.h5
```

```
if 'SAVE-TRAINED-MODEL' and False:  # Remove 'and False' to save the model.
  cnn_model.save('mnist_cnn_model.h5')
  !ls -la mnist_cnn_model.*
  cnn_converter = tf.lite.TFLiteConverter.from_keras_model(cnn_model)
  tflite_cnn_model = cnn_converter.convert()
  #
  # Save the model.
  with open('mnist_cnn_model.tflite', 'wb') as f:
    f.write(tflite_cnn_model)
  #
  files.download('mnist_cnn_model.h5')
  files.download('mnist_cnn_model.tflite')


if 'UPLOAD-MODEL' and True:  # Upload a model instead.
  model_data = next(iter(files.upload().values()))
  with open('mnist_cnn_model.h5', 'wb') as h_model:
    h_model.write(model_data)
  reloaded_cnn_model = tf.keras.models.load_model("mnist_cnn_model.h5")
  #
  print(
      f'Test set accuracy: {reloaded_cnn_model.evaluate(ds_test)[1] * 100:.2f}%'
      )
```

```
313/313 [==============================] - 5s 15ms/step - loss: 0.0244 -
sparse_categorical_accuracy: 0.9917
Test set accuracy: 99.17%
```

17 Other major architectural ideas

We discussed convolutional networks for image recognition tasks, and also some of the common approaches to improve model performance (which typically means: generalization).

Our next major focus will be "using the TensorFlow machinery to do Physics with it - perhaps without any ML involved". So, this may be a good opportunity to discuss a few other important general ML-related ideas.

17.1 Embeddings

Much of "supervised Machine Learning" (i.e. we have target labels) typically is about:

1. Representing some aspect(s) of the real world we care about as a high-dimensional vector. (This might be: an image, a molecular structure, a sentence, a medical record, etc.)
2. Defining a "model" $m_{\vec{\theta}}$ with trainable parameters $\vec{\theta} \in \mathbb{R}^D$ that can produce the desired output.
3. Obtaining training examples and tweaking model parameters (typically via some variant of stochastic gradient descent) to handle training set examples well.
4. Measuring performance on a validation-set.

5. Once everything looks good, measuring how well the model generalizes on the test set, and deploying the model.

We want to focus on 1.: How do we turn something like a sentence into a vector?

The "Netflix Prize Problem" provides an interesting setting to study this question. The Wikipedia article has details about the back story, but in a nutshell, this was about the Netflix video streaming company setting up a $1M competition back in 2006 for building a video recommendation system that outperforms their own system by at least 10%.

Academically, what was interesting about this problem was that Netflix provided a very large training dataset. Getting training data labels is generally expensive, and here the research community had an opportunity to use a dataset much larger than what they normally had available, at the order of 100M training examples.

The problem was as follows: Netflix customers watch movies, and get an opportunity to rate how well they enjoyed a movie right after they watched it. These ratings do obviously contain personal preferences about movies, and so it clearly is attractive for Netflix to have a recommendation system that produces individualized suggestions about what other movies a customer might enjoy watching.

For our purposes, we can imagine the overall setting to be as follows: We have a large "user movie ratings" matrix R_{um}, where user u rates movie m - let's say with score $R_{um} \in [-1..1] \cup \mathrm{NaN}$, where NaN is supposed to mean: "this user has not rated (perhaps not watched) that movie". We might have additional information about movies, but here, we want to focus exclusively on this matrix.

The matrix that we are given has some entries blanked out - Netflix knows how the given user rated the given movie, but for some (perhaps a million or so) ratings, they do not tell us and instead have put a NaN entry there, just as if the user had not yet rated the movie. We do not know which entries are the missing ones, but we want to build a system that can predict these ratings well.

Let us have a break here and give everybody 10 minutes to think about the problem.

In the end, the prize was won in 2009, in a "photo finish". The winning team did a hundred different things, but we want to focus on the core idea.

This was simply to try to find two matrices U, M such that $R \approx UM$ holds for the known entries. With indices, $R_{um} = \tilde{U}_{uk}\, \tilde{M}_{km}$. Here, u is still a user-index (going up to perhaps 100 000), and m is still a movie index (going up to maybe 10 000 or so). The range of the index k is small-ish, perhaps going up to $K = 50$.

We see how we can regard this as an optimization problem. We also immediately see how we can use such a factorization to make predictions. But why is doing this useful?

Suppose individual customers have movie preferences that could be described as "generally likes Jackie Chan movies", "likes comedy", "dislikes horror movies". Now, if we had a fixed list of perhaps 200 such categories, we might try to get to a quantitative prediction if we find every user's and every movie's "profile" as a vector of quantitified alignment with each category. Then, the alignment between a given user's and movie's profile should allow us to predict an unknown rating.

The beauty of this "matrix factorization" approach is that it determines these categories for us, in such a way that they are most useful for the problem at hand!

Now, of course, as stated, the problem has a $GL(K)$ ambiguity, since we can always transform $\tilde{U} \to \tilde{U}\Lambda$, $\tilde{M} \to \Lambda^{-1}\tilde{M}$, with $\Lambda \in GL(K)$. But let's say we deal with this somehow, perhaps by

additionally postulating that every row-vector of $\tilde{U}$ and every column-vector of $\tilde{M}$ must be length-1. This would then still leave us with some ambiguity (we could still do an $O(K)$ rotation of the coordinate basis), but apart from such details, we are left with a linear preference model where the problem itself seeks out the relevant directions.

Another way to view this problem: If all entries of R were known, it very likely would not be a "random" matrix but have extra structure. So, the question is: if we wanted to do data reduction and store far fewer elements, what would be a good proposal to still get a reasonably good approximation to the original matrix? Intuitively, "the best thing we can usually do" is to perform a Principal Component Analysis. This amounts to writing the matrix as a product $R = USV^T$ where here, since R is real, U and V are orthogonal, and S is rectangular with entries only on the diagonal~$S_{(j)(j)}$ - and then trimming S by only retaining the K largest-magnitude "generalized eigenvalues". So, the approach is: "Perform a Singular Value Decomposition, and project out all smallish generalized eigenvalues". This is, of course, a tried and tested pre-Deep-Learning ML approach that is widely used in many disciplines, also in financial analysis.

Can we then interpret these "principal axes"? It turns out that, indeed, if we work this out for the Netflix matrix, we find that the two most relevant dimensions are "Drama-vs-Comedy" and "Unsurprising-Plot-vs-Plot-Twists".

A nicely readable article about this is: Matrix Factorization Techniques for Recommender Systems.

Now, what does this mean for Deep Learning? A basic insight is that if we have tokenized input (such as for text: each word or multi-word term in the dictionary is one token), we often can simply map each token to some K-dimensional vector with a trainable mapping (which we call an "embedding"), and leave it to the problem at hand to adjust the token-to-vector mapping in a way that maximizes usefulness-for-the-problem.

This approach has produced some interesting surprises. Leaving out some (not quite irrelevant) detail, training a word embedding model on a problem that involves examples from news articles, for example, one may well find that the embedding evolved some lattice-like structure. If $E : \text{token} \to \mathbb{R}^K$ is the trained token-embedding function, we may find that equations such as $E(\text{Paris}) - E(\text{France}) + E(\text{Germany}) \approx E(\text{Berlin})$ hold (in the sense that among the nearest neighbors of the left-hand-side vector, we find the embedding vector for "Berlin" on typically the 1st, but generally maybe at most 2nd or 3rd place).

Google put some code from 2013 online that allows exploring this phenomenon. It meanwhile has been affected by some moderate "bit rot", but for those who are curious, the web page is `https://code.google.com/archive/p/word2vec/`, and the code archive is: `https://storage.googleapis.com/google-code-archive-source/v2/code.google.com/word2vec/source-archive.zip`

17.2 The wider ML Landscape

Some important notions and ideas from the wider Machine Learning landscape that we at least should have mentioned, but will not have any time to go into detail about.

- Unsupervised Learning

  In general: Learning where we have no ground truth labels.

  - Autoencoders (Variations on the "input → {compact representation} → original" idea.)

  - t-SNE as a "useful heuristic for visualizing in 2d or 3d how data clusters in high dimensions".

- Semi-Supervised Learning

  Learning where we have some labeled examples, and many more unlabeled examples. Idea: "Knowing how stuff generally looks like will typically help".

- Reinforcement Learning

  Learning behavior from reward observations.

- Regression Problems

  Estimating a non-discrete quantity. Key questions revolve around "how much spread is there in the 'labels'", and 'what features allow me to explain how much of that spread?' I.e. "Standard deviation of the height of all children attending primary school is X, but if I know their age, I can predict their height with standard deviation $Y < X$.

- Non-Neural-Network Approaches
  - K-Means Classifiers
  - Support Vector Machines and Kernel Machines

- Specialized NN architectures
  - Sequence-to-Sequence learning: LSTM ('ancient' but quite powerful).
  - Graph-NNs(/-CNNs)
  - "Transformer Architectures" More recent, quite powerful, we are still in the process of properly understanding what they can do and how they work. Basis for large language models such as GPT-3.
  - Generative Adversarial Networks - for generating e.g. realistic-looking art.

    Also, for example: "Neural Photo Editing" (Example YouTube video - paper [1]).

- Tweaking pre-trained models; TensorFlow Hub

18 Using TensorFlow for Physics

We will now, for the time being, leave Machine Learning behind and explore the deeper levels of TensorFlow, the fast-gradients-on-GPU-and-TPU machinery, and see how we can use this to our advantage when doing physics.

18.1 Accelerating computations with Graphics Hardware

Comparing "how our numpy-based and TensorFlow-based MNIST code feel like", I think it is fair to say that "TensorFlow is a fast beast".

When we run on GPU, the rules of the game are that the GPU contains many small processor-like units, which however cannot do too complex control flow operations, and work in groups where in each group, all must perform the same steps (generally going through different data). Bit like rowers on multiple rowing eights.

Want to do tensor arithmetics in such a way that we can put the computation onto a CPU, or alternatively onto such an architecture. It helps to understand the overall design if we first look into earlier ideas and approaches towards "making some computations execute either on CPU or on (say, even pre-GPU) graphics hardware". The first such applications were, unsurprisingly, about making computer graphics fast.

The main abstraction that evolved (first in Silicon Graphics's "IRIS GL" system, which became the OpenGL standard) was that of a "display list". A "display list" basically is a sequence of 3d graphics related drawing commands such as "draw a triangle with these vertex 3d coordinates" which form a unit that can either be processed by code running on CPU, or more directly by silicon in the graphics processing unit.

In principle, we can do the following thing:

1. Tell the OpenGL library to start recording all subsequent drawing commands on a display list.
2. Go through a lot of complicated code which might do things such as running a fractal generator to generate some realistic-looking landscape on-demand, but also will - interspersed in that calculation - at some point execute 3d drawing commands.
3. Tell the OpenGL library to stop recording and hand back the sequence of recorded drawing commands as a "display list object".
4. Perhaps change perspective, lighting, or other details, and re-execute the recorded list of 3d drawing commands to re-render the scene, perhaps from another viewpoint.

Pretty much the same strategy also would work for recording and redoing tensor arithmetics, if we made all tensor-arithmetics operations use a library that can switch into "recording mode". The question is: Why would one want to do that? With 3d rendering, it's clear that we might want to redo rendering a given scene from a different perspective. But what would we want to do differently when redoing a tensor arithmetic calculation? After all, if we did feed slightly different input data, chances are that we might branch differently at some point, and the actual calculation would then follow a path that is not the one recorded on the tape. Still, there actually is much benefit in having such a tensor-arithmetic code, just because it has all the go-left-or-right decisions baked in that have been made "on the forward calculation". This is straight code, without branching, that tells us what numerics operations actually have happened, and so we would have a very easy time doing sensitivity-backpropagation on this.

But let us take an actual look at an OpenGL display list, to show that this is no dark magic.

[ ]:
```
!pip install pyopengl
# One issue with `pyopengl` is that unless GLUT and GLU libraries
# are installed, this will fail with cryptic error messages, so we need
# to ensure we have GLUT and GLU packages - plus an X virtual framebuffer
# to render X in headless mode.
!apt install xvfb freeglut3 libglu1-mesa
# Starting a virtual-framebuffer X server and setting this as our default
# X11 display.
!nohup >/dev/null 2>&1 Xvfb :7 -screen 0 1024x1024x24 &
import os
os.environ['DISPLAY'] = ':7.0'
```

```
# Some additional modules and extensions used further down in this notebook.
import collections
import datetime
import itertools
import math
import sys

from PIL import Image
from PIL import ImageOps

from OpenGL import GL as gl
from OpenGL import GLU as glu
from OpenGL import GLUT as glut

import scipy.interpolate
import scipy.optimize

import numpy
import tensorflow as tf

from matplotlib import pyplot
from matplotlib import image as mpimage

# Load the TensorBoard notebook extension.
%load_ext tensorboard
```

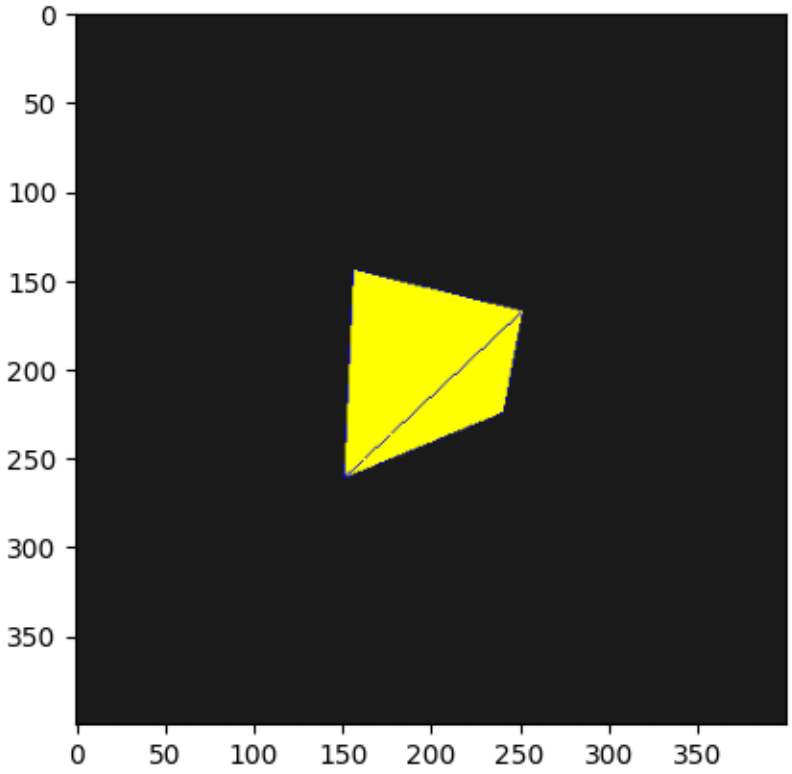
```
Requirement already satisfied: pyopengl in /usr/local/lib/python3.10/dist-
packages (3.1.7)
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
libglu1-mesa is already the newest version (9.0.2-1).
freeglut3 is already the newest version (2.8.1-6).
xvfb is already the newest version (2:21.1.4-2ubuntu1.7~22.04.1).
0 upgraded, 0 newly installed, 0 to remove and 16 not upgraded.
The tensorboard extension is already loaded. To reload it, use:
  %reload_ext tensorboard
```

[ ]:
```
def opengl_demo():
  gl_delete_lists_calls = []
  #
  def setup_display_lists():
    gl_list_start = gl.glGenLists(1)
    # BEGIN optimized display-list for rendering a tetrahedron.
    gl.glNewList(gl_list_start, gl.GL_COMPILE)
    gl.glBegin(gl.GL_TRIANGLES)
    #
```

```
    gl.glVertex3f(+2.0, -2.0, -2.0)
    gl.glVertex3f(-2.0, +2.0, -2.0)
    gl.glVertex3f(-2.0, -2.0, +2.0)
    #
    gl.glVertex3f(+2.0, +2.0, +2.0)
    gl.glVertex3f(-2.0, -2.0, +2.0)
    gl.glVertex3f(-2.0, +2.0, -2.0)
    #
    gl.glVertex3f(+2.0, -2.0, -2.0)
    gl.glVertex3f(+2.0, +2.0, +2.0)
    gl.glVertex3f(-2.0, -2.0, +2.0)
    #
    gl.glVertex3f(+2.0, -2.0, -2.0)
    gl.glVertex3f(-2.0, +2.0, -2.0)
    gl.glVertex3f(+2.0, +2.0, +2.0)
    #
    gl.glEnd()
    # BEGIN optimized display-list for rendering a tetrahedron.
    gl.glEndList()
    #
    gl_delete_lists_calls.append(lambda: gl.glDeleteLists(gl_list_start, 1))
    return {'tetrahedron': gl_list_start,
            # We could have further display lists here.
            }
#
width, height = 400, 400
glut.glutInit()
glut.glutInitDisplayMode(glut.GLUT_DOUBLE | glut.GLUT_RGB | glut.GLUT_DEPTH)
glut.glutInitWindowSize(width, height)
glut.glutCreateWindow('GL')
glut.glutHideWindow()
gl.glClearColor(0.1, 0.1, 0.1, 1.0)
gl.glColor(0.0, 1.0, 0.0)
try:
    display_lists = setup_display_lists()
    def render():
        gl.glEnable(gl.GL_TEXTURE_2D)
        gl.glEnable(gl.GL_DEPTH_TEST)
        gl.glClearDepth(1.0)
        gl.glDepthFunc(gl.GL_LESS)
        gl.glDepthMask(gl.GL_TRUE)
        gl.glShadeModel(gl.GL_SMOOTH)
        gl.glViewport(0, 0, width, height)
        gl.glMatrixMode(gl.GL_PROJECTION)
        gl.glLoadIdentity()
        glu.gluPerspective(45.0, width / height, 0.1, 1000.0)
        gl.glMatrixMode(gl.GL_MODELVIEW)
```

```
      gl.glLoadIdentity()
      # Lighting
      gl.glLightfv(gl.GL_LIGHT0, gl.GL_AMBIENT, (0.5, 0.5, 0.5, 1.0))
      gl.glLightfv(gl.GL_LIGHT0, gl.GL_DIFFUSE, (1.0, 1.0, 1.0, 1.0))
      gl.glLightfv(gl.GL_LIGHT0, gl.GL_POSITION, (2.0,23.0, -20.0, 1.0))
      gl.glEnable(gl.GL_LIGHT0)
      glu.gluLookAt(0.0, 20.0, 7.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0)
      gl.glClear(gl.GL_COLOR_BUFFER_BIT | gl.GL_DEPTH_BUFFER_BIT)
      gl.glPolygonMode(gl.GL_FRONT_AND_BACK, gl.GL_FILL)
      gl.glColor3f(1.0, 1.0, 0.0)
      gl.glCallList(display_lists['tetrahedron'])
      gl.glColor3f(0.0, 0.0, 1.0)
      gl.glPolygonMode(gl.GL_FRONT_AND_BACK, gl.GL_LINE)
      gl.glCallList(display_lists['tetrahedron'])
    #
    render()
    glut.glutSwapBuffers()
    gl.glPixelStorei(gl.GL_PACK_ALIGNMENT, 1)
    data = gl.glReadPixels(0, 0,
                           width, height,
                           gl.GL_RGB, gl.GL_UNSIGNED_BYTE)
    image = Image.frombytes("RGB", (width, height), data)
    image.save('GL.png', 'PNG')
  finally:
    # Ensure deallocation of all display lists before we leave
    # this call frame.
    for f in gl_delete_lists_calls: f()


opengl_demo()
pyplot.imshow(mpimage.imread('GL.png'))
pyplot.show()
```



So... "is this how TensorFlow works"? Not quite - but this "arithmetic tape" approach to back-propagation naturally is popular for many such systems, and it sometimes shows in names such as "TAPENADE [10]" (which, incidentally, uses a somewhat more clever approach than a bare tape). The idea has however entered the TensorFlow2 API in the form of a "Gradient Tape".

So, what then does TensorFlow do? Any given intermediate (tensor) result in a tensor-arithmetic computation is part of a directed acyclic graph that represents the calculation, and the tensors "flow" along the edges of this graph, which show how the output of one operation feeds into other "downstream" operations.

Having this graph has many advantages: One may think that even with a graph, when we actually do the computation, we have to go through all nodes in some particular order - but that is actually not true on massively parallel hardware, where we can make decisions to have different parts of the graph evaluated concurrently, on different devices. So, given some particular computing architecture with parallel capabilites, there is an interesting problem around "how to make this particular tensor-arithmetic calculation fast on that specific hardware?" - and this is what TensorFlow (i.e. the lower architectural layers of TensorFlow) do for us. Also, of course, it understands sensitivity-backpropagation in terms of a graph-transformation.

Let's jump straight in and actually see an example. We will be using TensorFlow2 with TensorBoard for visualization. Overall, TensorBoard is more often used to visualize higher level things than function-graphs, but here this is appropriate. Let's do some very simple physics: computing the energy of N electrons placed on a unit sphere in 3d.

For simplicity, we use a redundant description where every electron is described by 3 coordinates, but we unit-normalize the vector to put them all on a sphere.

[ ]:
```
@tf.function
def thomson_energy(electron_coords):
  """Computes the electrostatic energy of electrons on a sphere.

  Args:
    electron_coords: [num_electrons, 3] float tf.Tensor.
      Per-electron 3d position. This gets projected onto the unit sphere before
      working out electrostatic energy.
  Returns:
    float tf.Tensor, the total electrostatic energy.
  """
  projected_electron_coords = (
      electron_coords / tf.linalg.norm(electron_coords, axis=-1, keepdims=True))
  # There is TensorFlow library code for computing a pairwise-distance matrix,
  # but here, let us show how to do that manually,
  # using reshaping and broadcasting.
  distance_vecs = (projected_electron_coords[:, tf.newaxis, :] -
                   projected_electron_coords[tf.newaxis, :, :])
  distances = tf.linalg.norm(distance_vecs, axis=-1)
  # We also need inverse-distances. The problem is that every electron has
  # distance zero to itself. We could use tf.math.divide_no_nan() here,
  # but then we would have some explaining to do how this works with
  # gradients, and why we can take it as a guarantee to indeed always see exact
  # zeros on the matrix diagonal, despite doing approximate numerical
  # calculations. Let us instead show another trick
  # (mostly to illustrate some techniques).
  # If, for example, num_electrons = 3, our distance matrix is of the form:
  #
  # [[d00, d01, d12],
  #  [d10, d11, d12],
  #  [d20, d21, d22]]
  #
  # The problematic elements are along the diagonal.
  # If we trim `d22` and reshape the remaining N^2-1 elements to [N-1, N+1],
  # we get this form:
  #
  # [[d00, d01, d02, d10],
  #  [d11, d12, d20, d21]]
  #
  # ...where we can just drop the initial column.
  num_electrons = tf.shape(electron_coords)[0]
  new_shape = tf.stack([num_electrons - 1, num_electrons + 1], axis=0)
  reshaped_dists = tf.reshape(tf.reshape(distances, [-1])[:-1], new_shape)
  return 0.5 * tf.math.reduce_sum(1 / reshaped_dists[:, 1:])
```

```
  # Participant Exercise: Use this alternative strategy instead:
  # - Adding a unit matrix to the distances-matrix
  # - Inverting and summing
  # - Subtracting the known contribution coming from the diagonal.


# Let's try this out and compute the energy of electrons in tetrahedral
# configuration.
#
# Here, we will always feed tf-Tensors to tf-functions, but having a numpy
# representation of input data is useful, since this simplifies some operations.
tetrahedral_arrangement = numpy.array(
    [[-1, -1, -1], [-1, 1, 1], [1, -1, 1], [1, 1, -1]],
    dtype=numpy.float64)

print('E(tetrahedral):',
      thomson_energy(tf.constant(tetrahedral_arrangement, dtype=tf.float64)))
# If we scale all coordinates, this should not affect the energy, since
# all electron positions get projected onto the unit sphere.

print('E(10*tetrahedral):',
      thomson_energy(tf.constant(10 * tetrahedral_arrangement,
                                 dtype=tf.float64)))

# Here, we got "eager-mode TensorFlow tensors" back.
# These objects have a .numpy() method which extracts numpy data:

numpy_energy =  thomson_energy(tf.constant(tetrahedral_arrangement,
                               dtype=tf.float64)).numpy()
print(repr(numpy_energy))


# Let us actually check that if we perturb the geometry a bit, the energy
# goes up.

def get_tetrahedral_energy_change_for_random_perturbation(pert):
  # Helpers
  tc = lambda x: tf.constant(x, dtype=tf.float64)
  # Strictly speaking, unit_normalized() is unnecessary here.
  e0 = thomson_energy(tc(tetrahedral_arrangement)).numpy()
  e1 = thomson_energy(tc(tetrahedral_arrangement + pert)).numpy()
  return e1 - e0

print(get_tetrahedral_energy_change_for_random_perturbation(
      numpy.random.RandomState(seed=1).normal(size=(4, 3),
                                              scale=1e-5)))
# Participant Exercise: Check that even with different RNG seeds,
```

```
# we will always see an increase in energy if we perturb the tetrahedral
# arrangement. Magnitude-wise, this increase looks as if it were
# "to 2nd order in epsilon".
```

```
E(tetrahedral): tf.Tensor(3.674234614174767, shape=(), dtype=float64)
E(10*tetrahedral): tf.Tensor(3.674234614174767, shape=(), dtype=float64)
3.674234614174767
1.542903582674171e-10
```

Let us try to get a look at the actual tensor arithmetics graph for this little computation. We will use the 'tensorboard' tool for that.

[ ]:
```
# The function to be traced.
#
# This is just the same computation we defined earlier, but with code comments
# removed and some extra "with tf.name_scope()" and constant-names added.
# The purpose of these is to provide more hierarchical structure for
# tensor-arithmetic graph visualization with TensorBoard.
@tf.function
def thomson_energy_for_visualization(electron_coords):
  with tf.name_scope('normalization'):
    projected_electron_coords = (
        electron_coords / tf.linalg.norm(electron_coords, axis=-1,
                                         keepdims=True))
  with tf.name_scope('distances'):
    distance_vecs = (projected_electron_coords[:, tf.newaxis, :] -
                     projected_electron_coords[tf.newaxis, :, :])
    distances = tf.linalg.norm(distance_vecs, axis=-1)
  with tf.name_scope('reshaping'):
    num_electrons = tf.shape(electron_coords)[0]
    new_shape = tf.stack([num_electrons - 1, num_electrons + 1], axis=0)
    reshaped_dists = tf.reshape(tf.reshape(distances, [-1])[:-1], new_shape)
  with tf.name_scope('energy'):
    return (tf.constant(0.5, dtype=tf.float64, name='1/2') *
            tf.math.reduce_sum(1 / reshaped_dists[:, 1:]))

# Set up logging.
!rm -rf logs
stamp = datetime.datetime.now().strftime("%Y%m%d-%H%M%S")
logdir = 'logs/func/%s' % stamp
writer = tf.summary.create_file_writer(logdir)

# Sample input data. Should be created ahead-of-tracing.
example_electron_coords = tf.random.uniform((5, 3), dtype=tf.float64)

# Bracketing the function call with
# tf.summary.trace_on() and tf.summary.trace_export().
tf.summary.trace_on(graph=True, profiler=True)
```

```
# Calling only one tf.function when tracing.
try:
  energy = thomson_energy_for_visualization(example_electron_coords)
  with writer.as_default():
    tf.summary.trace_export(
        name="demo_func_trace",
        step=0,
        profiler_outdir=logdir)
finally:
  tf.summary.trace_off()
```

```
WARNING:tensorflow:From /usr/local/lib/python3.10/dist-
packages/tensorflow/python/ops/summary_ops_v2.py:1332: start (from
tensorflow.python.eager.profiler) is deprecated and will be removed after
2020-07-01.
Instructions for updating:
use `tf.profiler.experimental.start` instead.
WARNING:tensorflow:From /usr/local/lib/python3.10/dist-
packages/tensorflow/python/ops/summary_ops_v2.py:1383: stop (from
tensorflow.python.eager.profiler) is deprecated and will be removed after
2020-07-01.
Instructions for updating:
use `tf.profiler.experimental.stop` instead.
WARNING:tensorflow:From /usr/local/lib/python3.10/dist-
packages/tensorflow/python/ops/summary_ops_v2.py:1383: save (from
tensorflow.python.eager.profiler) is deprecated and will be removed after
2020-07-01.
Instructions for updating:
`tf.python.eager.profiler` has deprecated, use `tf.profiler` instead.
WARNING:tensorflow:From /usr/local/lib/python3.10/dist-
packages/tensorflow/python/eager/profiler.py:150: maybe_create_event_file (from
tensorflow.python.eager.profiler) is deprecated and will be removed after
2020-07-01.
Instructions for updating:
`tf.python.eager.profiler` has deprecated, use `tf.profiler` instead.
```

[ ]:
```
%tensorboard --logdir logs/func
```

```
<IPython.core.display.Javascript object>
```

Let us next do something more interesting - using TensorFlow to turn our function into another function that computes the gradient of the energy.

Let's then also visualize the graph for that.

[ ]:
```
@tf.function
def thomson_energy_and_grad(electron_coords):
  tape = tf.GradientTape()
  with tape:
```

```
    tape.watch(electron_coords)
    energy = thomson_energy_for_visualization(electron_coords)
  return energy, tape.gradient(energy, electron_coords)


### Tensorboard graph visualization.
stamp = datetime.datetime.now().strftime("%Y%m%d-%H%M%S")
logdir = 'logs/grad/%s' % stamp
writer = tf.summary.create_file_writer(logdir)

tf.summary.trace_on(graph=True, profiler=True)
try:
  print(thomson_energy_and_grad(example_electron_coords))
  with writer.as_default():
    tf.summary.trace_export(
        name="demo_grad_trace",
        step=0,
        profiler_outdir=logdir)
finally:
  tf.summary.trace_off()

# The visualization may be a little bit messed up here, but we get the idea.
%tensorboard --logdir logs/grad
```

```
(<tf.Tensor: shape=(), dtype=float64, numpy=28.790527871655378>, <tf.Tensor:
shape=(5, 3), dtype=float64, numpy=
array([[ 12.91151948,  15.61737417, -18.2504733 ],
       [ -2.00207027, -11.13616609,   9.13746111],
       [ -7.64327007,   2.83715471,  49.863972  ],
       [ 23.05478418,  -9.21406972,  -9.10227686],
       [ -0.93899942,   9.80415117,  -3.73420366]])>)
```

```
<IPython.core.display.Javascript object>
```

18.2 TensorFlow 2 and Python

A `@tf.function` decorated function will get analyzed by TensorFlow2 and turned into some "callable GPU-ready graph-representation". This is a rather complex operation that could not be implemented if `tf.function` were a "normal" Python decorator which maps a black-box function to some other function that merely can call the original function. Rather, `@tf.function` has to somehow inspect the definition of the function at the code level.

With Lisp, such program-transformations would be naturally part of the language, and straightforward since the language basically has no syntax. This is much less the case with Python, where the corresponding program-transformation has to run a text parser and then analyze the parse tree.

One important caveat is that `@tf.function` decorated code might not execute as Python code, since the underlying "computation written in Python syntax" uses subtly different semantics from Python. This is most visibly when putting Python code with side effects under `@tf.function`:

```
[ ]: @tf.function
def tf2_execution_example1(x):
  print('CALLED with x:', x)
  return 2 * x

print('Step 1')
print(tf2_execution_example1(tf.constant(10, dtype=tf.float64)))
print('Step 2')
print(tf2_execution_example1(tf.constant(11, dtype=tf.float64)))
print('Step 3')
print(tf2_execution_example1(tf.constant(12, dtype=tf.float32)))
print('Step 4')
print(tf2_execution_example1(tf.constant(13, dtype=tf.float32)))
print('Step 5')
print(tf2_execution_example1(tf.constant(14, dtype=tf.float64)))
```

```
Step 1
CALLED with x: Tensor("x:0", shape=(), dtype=float64)
tf.Tensor(20.0, shape=(), dtype=float64)
Step 2
tf.Tensor(22.0, shape=(), dtype=float64)
Step 3
CALLED with x: Tensor("x:0", shape=(), dtype=float32)
tf.Tensor(24.0, shape=(), dtype=float32)
Step 4
tf.Tensor(26.0, shape=(), dtype=float32)
Step 5
tf.Tensor(28.0, shape=(), dtype=float64)
```

The noteworthy details here are:

1. In `Step 1`, when the `@tf.function` gets called, Python code that has side effects (the `print`) clearly gets executed.
2. The `tf.Tensor` object that gets printed apparently does not carry a value - at least, it does not report one. (This is not an "eager tensor".)
3. Re-evaluating the same function on different numerical input with the same shape and data types as used on an earlier evaluation re-uses the "compiled form" generated earlier - and since Python statement execution that does not somehow have a representation on the value dependency graph is invisible to this compiled form, statements with side effects are not executed here.
4. Evaluating the same decorated function with input of some other type (single-precision float rather than double-precision float) will produce another GPU-optimized graph, and in order to obtain that, Python code gets re-traced, and we get to see another side effect from a Python statement.
5. Compiled graphs are cached on the callable-function object produced by `@tf.function`, so if we switch back to using the previous data type, there already is a compiled representation of the graph available, and no re-tracing happens.

Let us compare this behavior with that of a not-`@tf.function` decorated equivalent function:

```
[ ]: def tf2_execution_example2(x):
       print('CALLED with x:', x)
       return 2 * x

     print('Step 1')
     print(tf2_execution_example2(tf.constant(10, dtype=tf.float64)))
     print('Step 2')
     print(tf2_execution_example2(tf.constant(11, dtype=tf.float64)))
     print('Step 3')
     print(tf2_execution_example2(tf.constant(12, dtype=tf.float32)))
     print('Step 4')
     print(tf2_execution_example2(tf.constant(13, dtype=tf.float32)))
     print('Step 5')
     print(tf2_execution_example2(tf.constant(14, dtype=tf.float64)))
```

```
Step 1
CALLED with x: tf.Tensor(10.0, shape=(), dtype=float64)
tf.Tensor(20.0, shape=(), dtype=float64)
Step 2
CALLED with x: tf.Tensor(11.0, shape=(), dtype=float64)
tf.Tensor(22.0, shape=(), dtype=float64)
Step 3
CALLED with x: tf.Tensor(12.0, shape=(), dtype=float32)
tf.Tensor(24.0, shape=(), dtype=float32)
Step 4
CALLED with x: tf.Tensor(13.0, shape=(), dtype=float32)
tf.Tensor(26.0, shape=(), dtype=float32)
Step 5
CALLED with x: tf.Tensor(14.0, shape=(), dtype=float64)
tf.Tensor(28.0, shape=(), dtype=float64)
```

As expected, we observe two things here:

1. Side effects clearly happen at every call, just as we would expect it for Python code.
2. The tensors reported on the `CALLED with:` lines are shown with numerical values: `CALLED with x: tf.Tensor(10.0, shape=(), dtype=float64)`. This was not the case before. In this "eager" mode, and only there, a `tf.Tensor` has a `.numpy()` method which allows us to extract its data as a `numpy.ndarray`.

This Python-code-to-GPU-graph translator has some limitations. It tries to present itself to us as if it had full understanding of a rather complicated language, but it actually does not work like a proper compiler. A true compiler would merely inspect source code and would not have any need for "tracing an actual function-invocation".

A function decorated with `@tf.function` looks like Python code, can be parsed into a syntax tree by the Python parser, and often can even be executed by CPython, but the actual rules about executing such a function differ from Python. TensorFlow (unlike numpy) allows - for example - putting a `tf.Tensor` object as a conditional into an `if`-statement:

[ ]:
```
def tf2_execution_example3(x):
  if x < 0:
    return -x
  return x

print(tf2_execution_example3(tf.constant([-20], dtype=tf.float64)))

print(tf2_execution_example3(numpy.array([5.0])))
```

```
tf.Tensor([20.], shape=(1,), dtype=float64)
[5.]
```

However, if we `@tf.function`-decorate a function that does this, we can encounter situations one would normally consider impossible under generally accepted notions of valid execution semantics. The problem is that `@tf.function` needs to trace-execute code rather than producing compiler output from mere inspection (it does not understand the complex Python language standard well enough to do that), and whenever there are conditionals, the corresponding graph representation "needs to have translations for all branches". We see the consequence of this in the next example:

[ ]:
```
@tf.function
def boom(tensor1):
  c0 = tf.constant(0, dtype=tf.int32)
  if tensor1:
    if not tensor1:
       # !!! This is the logically impossible case !!!!
       assert False, 'Boom!'
       return c0
    else:
       return c0
  else:
    return c0

boom(tf.constant(True, dtype=tf.bool))
```

```
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
<ipython-input-9-91e0dc041dad> in <cell line: 14>()
     12     return c0
     13
---> 14 boom(tf.constant(True, dtype=tf.bool))

/usr/local/lib/python3.10/dist-packages/tensorflow/python/util/traceback_utils.py
 ↪in error_handler(*args, **kwargs)
    151     except Exception as e:
    152       filtered_tb = _process_traceback_frames(e.__traceback__)
--> 153       raise e.with_traceback(filtered_tb) from None
    154     finally:
    155       del filtered_tb
```

```
/tmp/__autograph_generated_filewnozn99i.py in tf__boom(tensor1)
     55                         do_return = False
     56                         raise
---> 57                 ag__.if_stmt(ag__.ld(tensor1), if_body_1, else_body_1,␣
 ↪get_state_1, set_state_1, ('do_return', 'retval_'), 2)
     58                 return fscope.ret(retval_, do_return)
     59         return tf__boom

/tmp/__autograph_generated_filewnozn99i.py in if_body_1()
     45                             do_return = False
     46                             raise
---> 47                     ag__.if_stmt(ag__.not_(ag__.ld(tensor1)), if_body,␣
 ↪else_body, get_state, set_state, ('do_return', 'retval_'), 2)
     48
     49                 def else_body_1():

/tmp/__autograph_generated_filewnozn99i.py in if_body()
     29                     def if_body():
     30                         nonlocal retval_, do_return
---> 31                         assert False, 'Boom!'
     32                         try:
     33                             do_return = True

AssertionError: in user code:

    File "<ipython-input-9-91e0dc041dad>", line 7, in boom  *
        assert False, 'Boom!'

    AssertionError: Boom!
```

When writing low-level TensorFlow code, the author generally follows these rules which help a lot to avoid problems:

1. Do not interactively redefine functions while some TF code is running. (Some versions of TensorFlow identify functions by source-location, and changing these live can have unintended consequences.)
2. Use TensorFlow functions such as `tf.cond()`, `tf.while_loop()`, etc. rather than putting tensors into Python-level control flow statements (even if that would work).
3. Never mix tensors and non-tensor quantities, such as by using function signatures that state that a function may accept either a numpy-ndarray or alternatively also a `tf.Tensor`. Always maintain clarity about what is a `tf.Tensor` and what not.
4. Things having tensor input or output should have very simple call signatures.
5. Be mindful that `@tf.function` decorated functions are stateful objects that hold on to cached compiled versions. Despite being callable and also (generally) functionally-pure, it often is

more appropriate to think about them as stateful objects rather than "pure functions".

6. Be aware that `tf.where(c, x, y)` is implemented as equivalent to:

```
def tf_where(c, x, y):
  c1 = tf.cast(tf.cast(c, tf.bool), x.dtype)
  return c1 * x + (1 - c1) * y
```

...which means that a `NaN`-gradient even on a not-taken branch will percolate as a `NaN`.

18.3 Some Physics

Let's actually do something with all this knowledge. Let's wire it all up such that we can use `scipy.optimize.fmin_bfgs` to determine the minimal-energy configuration of four electrons on a sphere.

[ ]:
```
def get_min_energy_config_n_electrons(num_electrons):
  x0 = numpy.random.RandomState(seed=0).normal(size=(num_electrons, 3))
  def as_tf_input(x):
    return tf.constant(x.reshape(x0.shape), dtype=tf.float64)
  num_call_to_f = 0
  def f(x):
    nonlocal num_call_to_f
    result = thomson_energy_for_visualization(as_tf_input(x)).numpy()
    if num_call_to_f % 5 == 0:
      print(f'E: {result:16.12f}')
    num_call_to_f += 1
    return result
  def fprime(x):
    _, t_grad = thomson_energy_and_grad(as_tf_input(x))
    return t_grad.numpy().ravel()
  opt_config = scipy.optimize.fmin_bfgs(
      f, x0.ravel(), fprime=fprime,
      gtol=1e-12, maxiter=10**5).reshape(x0.shape)
  # Return energy and unit-normalized vectors.
  return (f(opt_config),
          opt_config / numpy.linalg.norm(opt_config, axis=-1, keepdims=True))


opt_energy_8, opt_config_8 = get_min_energy_config_n_electrons(8)
distance_statistics = collections.Counter(
    numpy.linalg.norm(opt_config_8[:, numpy.newaxis, :] -
                      opt_config_8[numpy.newaxis, :, :], axis=-1).round(5)
                      .ravel())

print(sorted((d, n / 8) for d, n in distance_statistics.items()))
```

```
E:  43.741776338794
E:  19.847570989142
E:  19.684686490780
```

```
E:  19.681629891495
E:  19.681580175157
E:  19.681330781762
E:  19.678334046533
E:  19.675773843023
E:  19.675289007551
E:  19.675287861456
E:  19.675287861233
Warning: Desired error not necessarily achieved due to precision loss.
         Current function value: 19.675288
         Iterations: 50
         Function evaluations: 53
         Gradient evaluations: 53
[(0.0, 1.0), (1.17125, 2.0), (1.28769, 2.0), (1.65639, 1.0), (1.89689, 2.0)]
```

[ ]:
```
# Let's check how far we can take this.
opt_energy_20, opt_config_20 = get_min_energy_config_n_electrons(20)
# opt_energy_200, opt_config_200 = get_min_energy_config_n_electrons(200)
```

```
E: 189.071509811076
E: 154.619941615551
E: 151.392468902219
E: 151.188079702750
E: 151.053550913645
E: 150.974088557642
E: 150.938160338286
E: 150.895697768632
E: 150.885305939055
E: 150.883374417498
E: 150.883210810365
E: 150.882725336300
E: 150.881636086169
E: 150.881574033464
E: 150.881568539280
E: 150.881568337436
E: 150.881568333757
Warning: Desired error not necessarily achieved due to precision loss.
         Current function value: 150.881568
         Iterations: 78
         Function evaluations: 84
         Gradient evaluations: 84
```

[ ]:
```
# For 8 electrons, this printed:
# [(0.0, 1.0), (1.17125, 2.0), (1.28769, 2.0), (1.65639, 1.0), (1.89689, 2.0)]
# So, for any given electron,
#
# - there is one electron at zero-distance.
# - there are two nearest neighbors...
```

```
# - two next-nearest neighbors...
# - one 3rd-nearest-neighbor,
# - and two "farthest-neighbors".
#
# This is not compatible with electrons sitting on the corners of a cube,
# where there would be 3 nearest-neighbors. Did we do anything wrong?
# Only if we assumed that for 8 electrons, cubic arrangement had lowest-energy.
# What is the energy of a cubic arrangement?

import itertools

cube_corners = list(itertools.product(*[(-1, 1)]*3))
print(cube_corners)

print('E(cube):',
      thomson_energy_for_visualization(
          tf.constant(cube_corners, dtype=tf.float64)).numpy(),
      'E_min:', opt_energy_8)

# So... indeed, our minimal-energy configuration has lower energy as the one
# with cubic symmetry.
```

```
[(-1, -1, -1), (-1, -1, 1), (-1, 1, -1), (-1, 1, 1), (1, -1, -1), (1, -1, 1),
(1, 1, -1), (1, 1, 1)]
E(cube): 19.7407740737628 E_min: 19.67528786123276
```

[ ]:
```
# Incidentally... we still have OpenGL loaded. Let's make good use of it.
import scipy.spatial

delaunay = scipy.spatial.Delaunay(opt_config_8)
tetrahedra = delaunay.simplices

count_by_surface = collections.Counter(
    frozenset(tet[:n].tolist() + tet[n + 1:].tolist())
    for n in (0, 1, 2, 3) for tet in tetrahedra)

outer_faces = [sorted(face) for face, count in
               count_by_surface.items() if count == 1]
print(outer_faces)

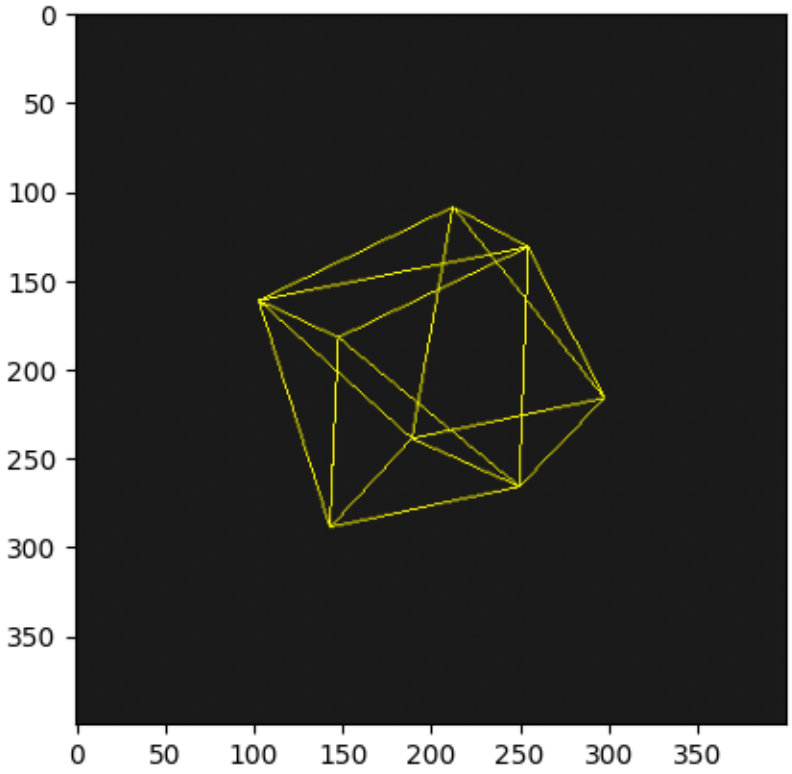

def gl_wireframe(filename, points, triangles):
  width, height = 400, 400
  glut.glutInit()
  glut.glutInitDisplayMode(glut.GLUT_DOUBLE | glut.GLUT_RGB)
  glut.glutInitWindowSize(width, height)
  glut.glutCreateWindow('GL')
```

```
  glut.glutHideWindow()
  gl.glClearColor(0.1, 0.1, 0.1, 1.0)
  gl.glViewport(0, 0, width, height)
  gl.glMatrixMode(gl.GL_PROJECTION)
  gl.glLoadIdentity()
  glu.gluPerspective(5.0, width / height, 0.1, 1000.0)
  gl.glMatrixMode(gl.GL_MODELVIEW)
  gl.glLoadIdentity()
  glu.gluLookAt(6.0, 40.0, 15.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0)
  gl.glClear(gl.GL_COLOR_BUFFER_BIT | gl.GL_DEPTH_BUFFER_BIT)
  gl.glPolygonMode(gl.GL_FRONT_AND_BACK, gl.GL_LINE)
  gl.glColor3f(1.0, 1.0, 0.0)
  gl.glBegin(gl.GL_TRIANGLES)
  for triangle in triangles:
    for num_point in triangle:
      gl.glVertex3f(*points[num_point])
  gl.glEnd()
  #
  glut.glutSwapBuffers()
  gl.glPixelStorei(gl.GL_PACK_ALIGNMENT, 1)
  data = gl.glReadPixels(0, 0,
                         width, height,
                         gl.GL_RGB, gl.GL_UNSIGNED_BYTE)
  image = Image.frombytes("RGB", (width, height), data)
  image.save(filename, 'PNG')


gl_wireframe('wire8.png', opt_config_8, outer_faces)


pyplot.imshow(mpimage.imread('wire8.png'))
pyplot.show()
```

```
[[3, 5, 7], [0, 3, 5], [5, 6, 7], [0, 1, 5], [4, 6, 7], [1, 5, 6], [3, 4, 7],
[0, 1, 2], [0, 2, 3], [1, 2, 6], [2, 4, 6], [2, 3, 4]]
```



The shape we get sort-of looks like a cube where we twisted the top square by 45 degrees relative to the bottom, and then allowed these squares to move a bit up/down on the sphere. Here, we render triangles, so these two squares are divided into two triangles. All the other faces are triangles.

Compare e.g. with: https://tracer.lcc.uma.es/problems/thomson/thomson.html

Participant Exercise: Work out corresponding shapes for more electrons. It might be helpful to tweak the energy-function with an extra spurious term that punishes electrons moving in 3d to have distance-from-origin-prior-to-normalization that is off of the desired value 1 by more than 1/2.

When writing such code, we might have to do some debugging. Some general tips around that:

- `@tf.function` decorated functions will be processed by `autograph` and have their logic turned into TensorFlow graphs that get executed instead of the Python code after first use (there are some complicated rules around this). When debugging, it often makes sense to instead stick with pure Python code manipulating "eager tensors" (basically, TF-wrapped numpy arrays). This achieves these things:
  - `print()` statements will get executed on every invocation.
  - We can actually call the `.numpy()` method on tensor-objects. (For code that gets autograph-translated, tf-tensors are "non-eager tensors", so "abstract elements on a graph" which do not have an associated numerical value.)

  - We can use the Python debugger (via `pdb.pm()` - details on how to do this in Colab) to inspect stacktraces.

- One somewhat convenient way to "have it both ways" is this:

```
import os

_DEFAULT_USE_AUTOGRAPH = 0

# Identity-decorator if env var `TF_USE_AUTOGRAPH` is unset and
# default is to not use it, or if `TF_USE_AUTOGRAPH` is set to
# anything else but '1'. Otherwise, equals tf.function.
maybe_tf_function = (
  tf.function if os.getenv('TF_USE_AUTOGRAPH',
                           str(_DEFAULT_USE_AUTOGRAPH)) == '1'
  else lambda f: f)

...

@maybe_tf_function
def my_function(...):
  ...
```

18.4 Rendering a Black Hole with TensorFlow

Since we now have all the bits and pieces together, let's actually build a TensorFlow-powered geodesic renderer for arbitrary Riemannian geometries - as an application, we want to do some basic raytracer-style rendering of a black hole.

This is a somewhat advanced exercise in "wiring up low-level TensorFlow in a more complicated calculation", which showcases many important techniques.

While this example in some sense is an abuse of TensorFlow, since this framework was not built to efficiently handle "small" tensors and shuttle them back and forth a lot between `numpy.ndarray` and `tf.Tensor`, it nevertheless allows us to discuss many interesting aspects. If one wanted to go for best possible achievable computational performance, this might not be a good approach, but as an illustration for how ML tools offer us new pathways to quickly do exploration on things that would be considerable effort without them, this certainly is interesting.

The example we are discussing in this section works in two directions - physicists who are familiar with the theory will be able to hold on to their theoretical knowledge and see the example as an illustration for how to accomplish rather nontrivial things with libraries that can also power Machine Learning. In the other direction, Machine Learning practitioners will be able to understand the wiring and hopefully be able to get a glimpse of how physics works in curved spacetime by studying a very concrete example in a way that is fully spelled out, with explicit numbers. In a way, our approach here allows course participants to experience the physics of black holes in yet another way that does not involve having to map complicated analytic expressions to geometric ideas.

Overall, if we wanted simply to render some specific geometry, such as Schwarzschild spacetime, there of course would be extra tricks available that should be exploited. There indeed by now are

interactive WebGL demos that allow live rendering of motion in a black hole geometry directly in the browser - such as this one by ESA.

Our aspirations here are different - we want to start from a simple `@tf.function` that computationally describes the local distance-function, then use TensorFlow to backpropagate this twice, in order to find the spacetime metric. Then, we backpropagate once more to get the Christoffel symbols. Next, we stick this into a very simple ODE-solver that can trace out batches of geodesics and think about how to then use this to render some objects.

Overall, some things are noteworthy about this approach:

- We will backpropagate through numerical matrix inversion or linear-equation-system-solving (since we use the inverse metric). This may be doubtful to do in general, since in many situations where we have a good "analytic formula" description of the metric, working out the analytic expression may be substantually less effort.
- We have seen that sensitivity-backpropagation is generally useful for 5+ parameters. Here, with only four spacetime coordinates, forward-mode AD might actually be a better choice. So, this is more about a "finger exercise" to show how we can wire up something like this rather than the overall best possible design.
- In order to make good use of the silicon on a GPU, we will process a collection of geodesics all at once. So, we generally want our tensor-arithmetic operations to be "batched". We have seen batch-indices in earlier ML related examples.

  Here, the presence of batch-indices leads to a strange interplay between the notion of "Einstein Summation" generally used in mathematics, and the (at first doubtful-looking) generalization widely used in Machine Learning.

  The rules of the game are codified in the behavior of the `numpy.einsum()` and `tf.einsum()` functions: for a summation prescription such as 'ab,ac->bc', the rule is: "Each combination of right hand side indices specifies a cell. All left hand side indices not present on the right hand side get summed over." This means that we can specify broken-looking "Einstein Summations" such as `numpy.einsum('i->', vector)` to get the sum of elements of a vector, or `numpy.einsum('ba,ba->b', vecs1, vecs2)` to compute a batch of scalar products for two batches of vectors (the batch-index being `b` here). Even more, there is special notation in such "generalized Einstein summation" for "a group of batch indices": `numpy.einsum('...a,...a->...', m, m)` would compute a batched-scalar-product where we may even have, say, two batch-indices, such as for "image-row" and "image-column", treating a rectangle of pixels as a doubly-indexed batch.

[ ]:
```
# Below, "tfb_" is used as a short-hand for a batch-tf.Tensor-to-batch-tf.Tensor
# graph-compiled TensorFlow function.

# Helper.
def tff64(x):
  return tf.constant(x, dtype=tf.float64)


@tf.function
def tfb_minkowski_ds2(tb_pos, tb_d_txyz):
```

```
  """Batched TensorFlow Minkowski Spacetime local scalar product."""
  return tf.einsum('...a,...a,a->...',
                   tb_d_txyz,
                   tb_d_txyz,
                   tff64([-1, 1, 1, 1]))


# Let's see how this works... Batch-scalar-product of a batch of 3 vectors,
# the first timelike, the second spacelike, the third null.
print(tfb_minkowski_ds2(
    tff64([[5, 6, 7]]),  # Arbitrary position.
    tff64([[1, 0, 0, 0],
           [0, 1, 0, 0],
           [1, 1, 0, 0]])).numpy())
```

```
[-1.  1.  0.]
```

[ ]:
```
# Let us define Schwarzschild geometry. Slight trick here: Should we ever
# evaluate this for r <= M, we make the result "Not-a-Number", as a canary for
# "this calculation accidentally entered forbidden territory due to
# limited-accuracy effects."

@tf.function
def tfb_schwarzschild_ds2(tb_pos, tb_d_txyz):
  """Batched TensorFlow Schwarzschild Spacetime local scalar product."""
  r_s = tff64(1)  # Let's use R_Schwarzschild = 1 here.
  tb_pos_xyz = tb_pos[..., 1:]
  tb_r = tf.linalg.norm(tb_pos_xyz, axis=-1)
  # We need the radial unit vector.
  tb_e_r = tb_pos_xyz / tb_r[..., tf.newaxis]
  tb_dr = tf.einsum('...i,...i,...k->...k', tb_d_txyz[..., 1:], tb_e_r, tb_e_r)
  db_d_rperp = tb_d_txyz[..., 1:] - tb_dr # "perpendicular-to-radial".
  # Geometry is simple in spacelike-perpendicular direction.
  ds2_rperp = tf.einsum('...i,...i->...', db_d_rperp, db_d_rperp)
  # The "Schwarzschild factor"
  s_factor = tf.where(tb_r > r_s,
                      1 - r_s / tb_r,
                      tff64(numpy.nan))
  ds2_radial = tf.einsum('...i,...i,...->...', tb_dr, tb_dr, 1 / s_factor)
  ds2_time = -tf.einsum('...,...,...->...',
                        tb_d_txyz[..., 0], tb_d_txyz[..., 0], s_factor)
  return ds2_time + ds2_radial + ds2_rperp


# Let's see how this works... Batch-scalar-product of a batch of 3 vectors,
# the first timelike, the second spacelike, the third null.
print('=== Batch of x-directed line-elements at increasing x-coordinate. ===')
```

```
print(tfb_schwarzschild_ds2(
    tff64([[0, 1.5, 0, 0],
           [0, 5, 0, 0],
           [0, 10, 0, 0],
           [0, 1000, 0, 0],
           [0, 1.01, 0, 0],  # Just above R_s
           [0, 0.1, 0, 0],   # Inside the black hole.
           ]),
    tff64([[0, 1, 0, 0]])).numpy())

print('=== Batch of y-directed line-elements at increasing x-coordinate. ===')
print(tfb_schwarzschild_ds2(
    tff64([[0, 1.5, 0, 0],
           [0, 5, 0, 0],
           [0, 10, 0, 0],
           [0, 1000, 0, 0],
           [0, 1.01, 0, 0],  # Just above R_s
           [0, 0.1, 0, 0],   # Inside the black hole.
           ]),
    tff64([[0, 0, 1, 0]])).numpy())

print('=== Batch of t-directed line-elements at increasing x-coordinate. ===')
print(tfb_schwarzschild_ds2(
    tff64([[0, 1.5, 0, 0],
           [0, 5, 0, 0],
           [0, 10, 0, 0],
           [0, 1000, 0, 0],
           [0, 1.01, 0, 0],  # Just above R_s
           [0, 0.1, 0, 0],   # Inside the black hole.
           ]),
    tff64([[1, 0, 0, 0]])).numpy())
```

```
=== Batch of x-directed line-elements at increasing x-coordinate. ===
[  3.          1.25        1.11111111   1.001001   101.
         nan]
=== Batch of y-directed line-elements at increasing x-coordinate. ===
[ 1.  1.  1.  1.  1. nan]
=== Batch of t-directed line-elements at increasing x-coordinate. ===
[-0.33333333 -0.8        -0.9        -0.999      -0.00990099         nan]
```

Note that this last scalar-product-of-t-directed-line-elements-with-themselves tells us that, for one unit of coordinate-time passing, there is the less physical time that is passing the closer we are to the black hole. (We of course need to take the square-root-of-magnitude of these numbers here, since we here see the "dT^2".)

Our next task is to obtain a "metric field", given the spacetime-position-dependent distance-squared. This is doable, but overall quite a bit messy.

Likely, we would be almost always better off here handling this step symbolically, but let us never-

theless see how this can be accomplished.

In general, we would here want to ask the user to provide a hand-coded metric-field-function.

[ ]:
```
_DEFAULT_USE_TF_FUNCTION = True


def maybe_tf_function(use_tf_function):
  return tf.function if use_tf_function else lambda x: x


# Slight wart: This function only allows a single batch-index.
# This is not much of a restriction, since we can wrap functions up
# into reshaping-to-one-batch-index-and-back.
def tfb_metric_fn(tfb_ds2, use_tf_function=_DEFAULT_USE_TF_FUNCTION):
  """Given a batched-ds2 function, return a batched-metric function."""
  # The metric is half the Hessian of the ds2-function.
  uc_zero = tf.UnconnectedGradients.ZERO
  @maybe_tf_function(use_tf_function)
  def tfb_metric(tb_pos):
    # First, we compute the ds2 for a zero-displacement ds=0.
    # We of course know this to be zero, but what matters is the
    # functional dependency, i.e. if we twice-backprop this to get
    # the Hessian, we obtain the metric.
    tcb_ds0 = tf.zeros_like(tb_pos)
    tape1 = tf.GradientTape()
    with tape1:
      tape1.watch(tcb_ds0)
      tape0 = tf.GradientTape(persistent=True)
      with tape0:
        tape0.watch(tcb_ds0)
        # Pretend tb_ds2 is batched and 1d-vector-valued,
        # since we need to take a Jacobian.
        # Unfortunately, there is no tape.batch_gradient,
        # so we have to go via tape.batch_jacobian.
        tb_ds2 = tfb_ds2(tb_pos, tcb_ds0)[..., tf.newaxis]
      # We need to take a batched-gradient.
      # That is available via GradientTape.batch_jacobian.
      # The indexing then removes the scalar-as-a-1d-vector
      # dimension again.
      grad = tape0.batch_jacobian(
          tb_ds2, tcb_ds0,
          unconnected_gradients=uc_zero)[:, 0, :]
    tb_hessian = tape1.batch_jacobian(grad, tcb_ds0,
                                      unconnected_gradients=uc_zero)
    return 0.5 * tb_hessian
  return tfb_metric
```

```
###
tfb_schwarzschild_metric = tfb_metric_fn(tfb_schwarzschild_ds2)
print(tfb_schwarzschild_metric(tff64(
    # Batch of positions.
    [[0, 5, 0, 0],
     [10, 5, 0, 0],
     [0, 1000, 0, 0]])).numpy())
```

```
[[[-0.8       0.        0.        0.      ]
  [ 0.        1.25      0.        0.      ]
  [ 0.        0.        1.        0.      ]
  [ 0.        0.        0.        1.      ]]

 [[-0.8       0.        0.        0.      ]
  [ 0.        1.25      0.        0.      ]
  [ 0.        0.        1.        0.      ]
  [ 0.        0.        0.        1.      ]]

 [[-0.999     0.        0.        0.      ]
  [ 0.        1.001001  0.        0.      ]
  [ 0.        0.        1.        0.      ]
  [ 0.        0.        0.        1.      ]]]
```

Let us actually compare this with a hand-coded Schwarzschild metric.

[ ]:
```
@tf.function
def tfb_schwarzschild_metric_handcoded(tb_pos):
  """Batched TensorFlow Schwarzschild Spacetime local scalar product."""
  r_s = tff64(1)  # Let's use R_Schwarzschild = 1 here.
  tb_pos_xyz = tb_pos[..., 1:]
  tb_r = tf.linalg.norm(tb_pos_xyz, axis=-1)
  s_factor = tf.where(tb_r > r_s,
                      1 - r_s / tb_r,
                      tff64(numpy.nan))
  one = tf.where(tb_r > r_s, tff64(1), tff64(numpy.nan))
  return tf.linalg.diag(tf.stack([-s_factor, 1 / s_factor, one, one],
                                 axis=-1))

print(tfb_schwarzschild_metric_handcoded(tff64(
    # Batch of positions.
    [[0, 5, 0, 0],
     [10, 5, 0, 0],
     [0, 1000, 0, 0]])).numpy())
```

```
[[[-0.8       0.        0.        0.      ]
  [ 0.        1.25      0.        0.      ]
  [ 0.        0.        1.        0.      ]
```

```
 [ 0.        0.        0.        1.      ]]

[[-0.8       0.        0.        0.      ]
 [ 0.        1.25      0.        0.      ]
 [ 0.        0.        1.        0.      ]
 [ 0.        0.        0.        1.      ]]

[[-0.999     0.        0.        0.      ]
 [ 0.        1.001001  0.        0.      ]
 [ 0.        0.        1.        0.      ]
 [ 0.        0.        0.        1.      ]]]
```

Next, let us get batched Christoffel symbols from the batched metric-function.

[ ]:
```
# Again, due to use of tape.batch_jacobian(),
# we actually only can do one batch-index.
def tfb_christoffel_fn_v0(tfb_fn_g, use_tf_function=_DEFAULT_USE_TF_FUNCTION):
  @maybe_tf_function(use_tf_function)
  def tf_christoffel(tb_pos):
    tape = tf.GradientTape()
    with tape:
      tape.watch(tb_pos)
      tb_g_at_pos = tfb_fn_g(tb_pos)
    tb_dg_at_pos = tape.batch_jacobian(tb_g_at_pos, tb_pos)
    # This variant uses the "inverse of the metric tensor".
    # Now, our code is set up to produce a NaN matrix inside the black hole,
    # and tf.linalg.inv() may raise an exception when it encounters
    # a non-invertible matrix. This is fixed in the 2nd variant below.
    tb_ginv_at_pos = tf.linalg.inv(tb_g_at_pos)
    tb_dg3_at_pos = (- tf.einsum('...abc->...cab', tb_dg_at_pos)
                     + tf.einsum('...abc->...bca', tb_dg_at_pos)
                     + tf.einsum('...abc->...acb', tb_dg_at_pos))
    return 0.5 * tf.einsum('...ab,...bcd->...acd',
                           tb_ginv_at_pos,
                           tb_dg3_at_pos)
  return tf_christoffel


def tfb_christoffel_fn(tfb_fn_g, use_tf_function=_DEFAULT_USE_TF_FUNCTION):
  @maybe_tf_function(use_tf_function)
  def tf_christoffel(tb_pos):
    tape = tf.GradientTape()
    with tape:
      tape.watch(tb_pos)
      tb_g_at_pos = tfb_fn_g(tb_pos)
    tb_dg_at_pos = tape.batch_jacobian(tb_g_at_pos, tb_pos)
    tb_dg3_at_pos = (- tf.einsum('...abc->...cab', tb_dg_at_pos)
                     + tf.einsum('...abc->...bca', tb_dg_at_pos)
```

```
                    + tf.einsum('...abc->...acb', tb_dg_at_pos))
    # Use tf.linalg.solve() rather than matrix-inversion.
    # This works with batched matrices, so we will need to reshape the
    # Christoffel symbols of the 1st kind
    # (batch_size, 4, 4, 4) -> (batch_size, 4, 16)
    # and reshape back again afterwards.
    return 0.5 * tf.reshape(
        tf.linalg.solve(tb_g_at_pos, tf.reshape(tb_dg3_at_pos, (-1, 4, 16))),
        (-1, 4, 4, 4))
  return tf_christoffel

###
tfb_christoffel_schwarzschild = tfb_christoffel_fn(
    tfb_schwarzschild_metric_handcoded)

christoffel_examples = tfb_christoffel_schwarzschild(tff64(
    # Batch of positions.
    [[0, 5, 0, 0],
     [10, 5, 0, 0],
     [0, 1000, 0, 0]])).numpy()

# This produces lengthy output:
# print(christoffel_examples.round(6))
```

[ ]:
```
# Naturally, Christoffel symbols will have many zero-entries.
# So, let us also use a helper to pretty-format such tensors.
#
# This code is taken from the m-theory Python library published alongside
# the current author's "numerical supergravity" papers,
# see: https://github.com/google-research/google-research/blob/master/m_theory/
 ↪m_theory_lib/m_util.py

def tformat(array,
            name=None,
            d=None,
            filter=lambda x: abs(x) > 1e-8,  # pylint:disable=redefined-builtin
            format='%s',  # pylint:disable=redefined-builtin
            nmax=numpy.inf,
            ncols=120):
  """Formats a numpy-array in human readable table form."""
  # Leading row will be replaced if caller asked for a name-row.
  if d is not None:
    array = array.round(d)
  dim_widths = [
      max(1, int(math.ceil(math.log(dim + 1e-100, 10))))
      for dim in array.shape]
  format_str = '%s: %s' % (' '.join('%%%dd' % w for w in dim_widths), format)
```

```
  rows = []
  for indices in itertools.product(*[range(dim) for dim in array.shape]):
    v = array[indices]
    if filter(v):
      rows.append(format_str % (indices + (v,)))
  num_entries = len(rows)
  if num_entries > nmax:
    rows = rows[:nmax]
  if ncols is not None:
    width = max(map(len, rows)) if rows else 80
    num_cols = max(1, ncols // (3 + width))
    num_xrows = int(math.ceil(len(rows) / num_cols))
    padded = [('%%-%ds' % width) % s
              for s in rows + ['']  (num_cols  num_xrows - len(rows))]
    table = numpy.array(padded, dtype=object).reshape(num_cols, num_xrows).T
    xrows = [' | '.join(row) for row in table]
  else:
    xrows = rows
  if name is not None:
    return '\n'.join(
        ['=== %s, shape=%r, %d%s / %d non-small entries ===' % (
            name, array.shape,
            num_entries,
            '' if num_entries == len(rows) else ' (%d shown)' % len(rows),
            array.size)] +
        [r.strip() for r in xrows])
  return '\n'.join(r.strip() for r in xrows)


def tprint(array, sep=' ', end='\n', file=sys.stdout, **tformat_kwargs):
  """Prints a numpy array in human readable table form."""
  print(tformat(array, **tformat_kwargs), sep=sep, end=end, file=file)



tprint(christoffel_examples, d=10)
```

```
0 0 1 0: 0.05       | 0 1 1 1: -0.025     | 1 1 0 0: 0.016      | 2 0 1 0:
1.001e-06  | 2 1 1 1: -5.005e-07
0 1 0 0: 0.016      | 1 0 1 0: 0.05       | 1 1 1 1: -0.025     | 2 1 0 0:
4.995e-07  |
```

What do we see here?

- The batch-index-0 and batch-index-1 Christoffel symbols are the same, as is expected since "we merely moved forward in time".
- The batch-index-2 Christoffel symbols are small, as is expected "faraway from the Schwarzschild radius".

For batch-index-2, starting "at rest", i.e. with a time-directed-only trajectory, and moving along it, forward in time, we get a small radial contribution pulling us inward ($\ddot{\gamma} = -\Gamma\dot{\gamma}\dot{\gamma}$). Magnitude-wise, the force looks as if it were $\propto 1/r^2$.

Let us see if we can compute some first trajectories.

[ ]:
```
from matplotlib import pyplot

def rk4_ode(df_ds, x0, s, ds):
  """(Batched) Runge-Kutta RK4 for numerically solving an ODE."""
  # https://en.wikipedia.org/wiki/Runge%E2%80%93Kutta_methods
  f1 = df_ds(x0, s)
  f2 = df_ds(x0 + 0.5 * ds * f1, s + 0.5 * ds)
  f3 = df_ds(x0 + 0.5 * ds * f2, s + 0.5 * ds)
  f4 = df_ds(x0 + ds * f3, s + ds)
  return x0 + (ds / 6.0) * (f1 + 2 * f2 + 2 * f3 + f4)


def rewrite_2nd_order_as_1st_order(f_acceleration, debug=False):
  def d_by_ds_xvs(xvs):
    # `xvs` is the vector
    # (x0, ..., x[n-1], v0, ..., v[n-1], curve_parameter_s).
    dim = (xvs.shape[-1] - 1) // 2
    return numpy.concatenate([
              xvs[..., dim : 2 * dim],  # "d/dt x = v"
              f_acceleration(           # "d/dt v = a"
                  xvs[..., :dim],
                  xvs[..., dim : 2 * dim]),
              numpy.ones_like(xvs[..., -1:])],
              axis=-1)
  return d_by_ds_xvs


def get_trajectories(f_acceleration, x0s, v0s, ds, s0=0):
  x0s = numpy.asarray(x0s)
  v0s = numpy.asarray(v0s)
  dim = x0s.shape[-1]
  d_by_ds_xvs = rewrite_2nd_order_as_1st_order(f_acceleration)
  def ode_f(xs, s):
    del s  # Curve-parameter actually gets ignored here.
    return d_by_ds_xvs(xs)
  xvs_now = numpy.concatenate(
    [x0s, v0s, numpy.zeros(shape=x0s.shape[:-1] + (1,), dtype=x0s.dtype)],
    axis=-1)
  for num_step in itertools.count():
    s = s0 + num_step * ds
    xvs_now = rk4_ode(ode_f, xvs_now, s, ds)
    yield xvs_now[..., :dim], xvs_now[..., dim : 2 * dim], xvs_now[..., -1]
```

```
###

def get_schwarzschild_renderer():
  tf_christoffel = tfb_christoffel_schwarzschild
  def f_acceleration_einstein(xs, vs):
    # Christoffel symbols at current position.
    gamma_amn = tf_christoffel(tf.constant(xs, dtype=tf.float64)).numpy()
    return -numpy.einsum('...amn,...m,...n->...a', gamma_amn, vs, vs)
  #
  # The function below is a closure over f_acceleration_einstein,
  # so this will compile the TensorFlow machinery on first execution and then
  # re-use it on subsequent ones.
  def fn_renderer(start_txyz, start_d_txyz_ds, ds=0.1):
    return get_trajectories(
      f_acceleration_einstein,
      # All start at (t, x, y, z) = (0, 5, 0, 0)
      start_txyz, start_d_txyz_ds, ds)
  return fn_renderer

def demo1_schwarzschild_trajectories(fn_renderer,
                                     vys,
                                     ds=0.1, num_steps=1000,
                                     fn_style_by_num_traj=lambda _: '-k'
                                     ):
  trajectories = fn_renderer(
    # All start at (t, x, y, z) = (0, -5, 0, 0)
    [[0.0, -5.0, 0.0, 0.0] for _ in vys],
    # This is actually a bit off and does not give us length-squared -1
    # tangents to the geodesic. See below for a discussion.
    [[1.0 / (1 + vy*vy)**.5, 0.0, vy, 0.0] for vy in vys],
    ds)
  xs_vs_ss = list(itertools.islice(trajectories, num_steps))
  fig = pyplot.figure(figsize=(10, 10))
  ax = fig.gca()
  ax.grid()
  ax.set_aspect('equal')
  alphas = numpy.linspace(0, 2*numpy.pi, 201)
  ax.plot(numpy.cos(alphas), numpy.sin(alphas), '-k')
  for num_trajectory in range(len(vys)):
    ax.plot([xs[num_trajectory, 1] for xs, vs, ss in xs_vs_ss],
            [xs[num_trajectory, 2] for xs, vs, ss in xs_vs_ss],
            fn_style_by_num_traj(num_trajectory))
  fig.show()
  return xs_vs_ss
```

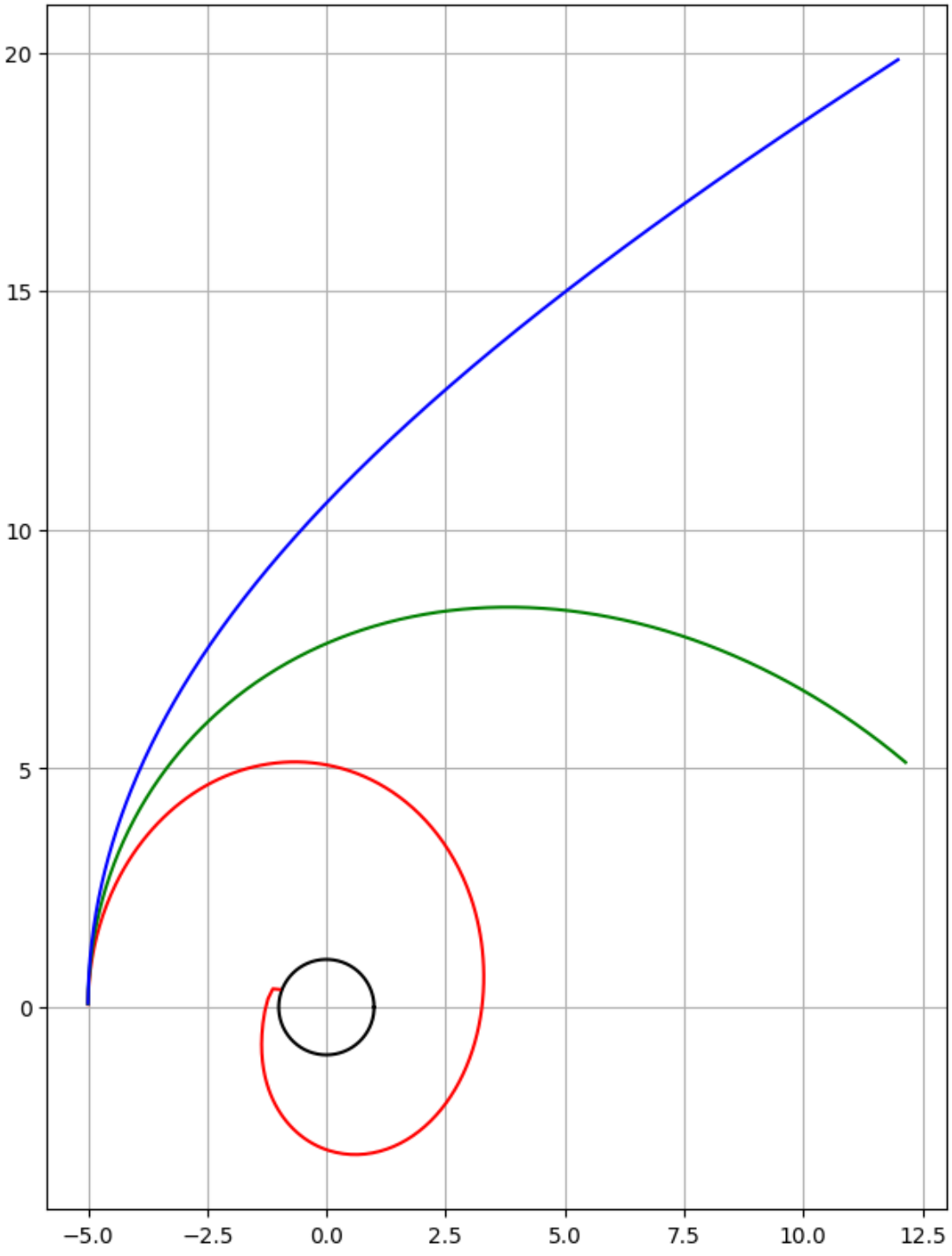
```
import time

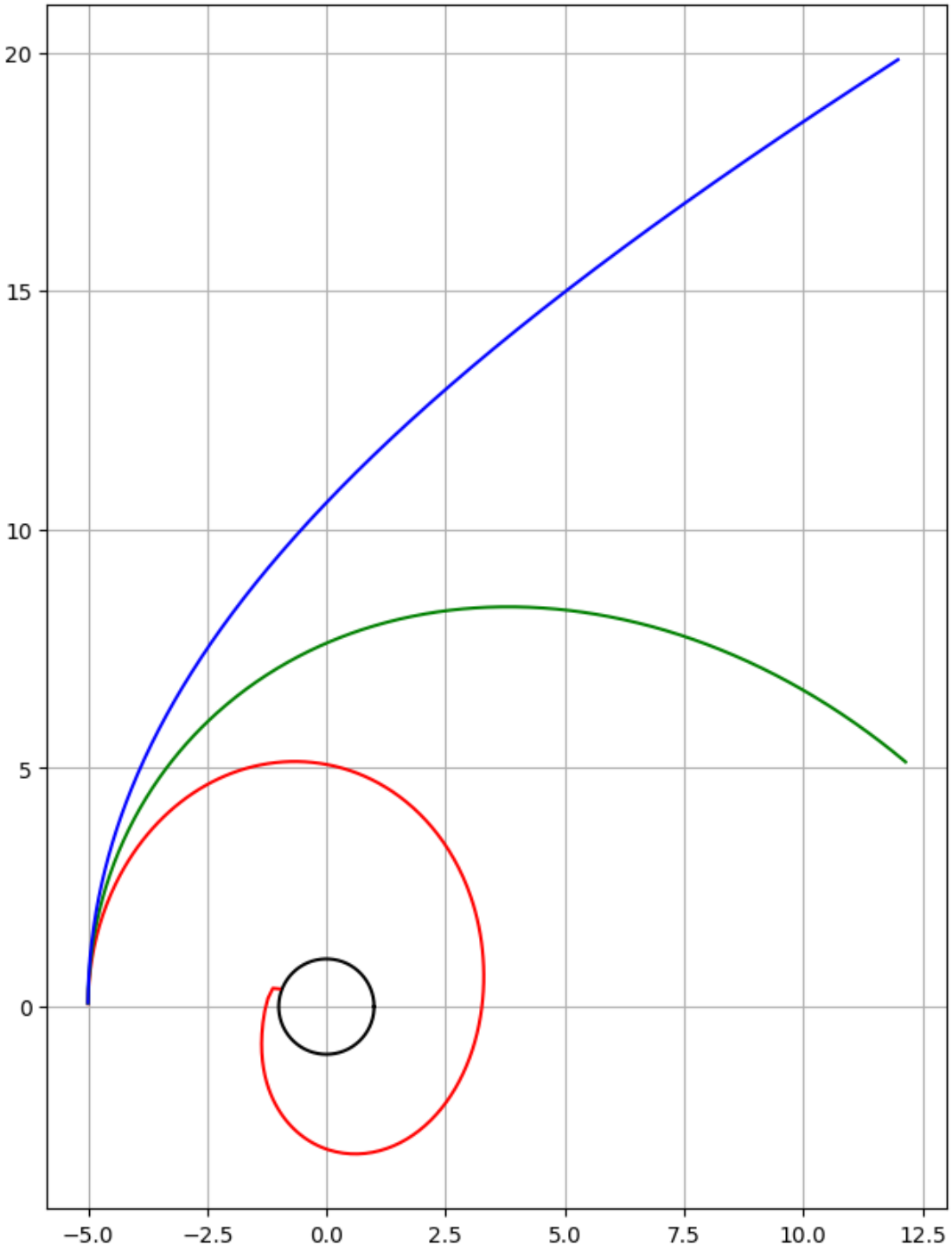
fn_renderer = get_schwarzschild_renderer()
t0 = time.time()
demo1_schwarzschild_trajectories(
    fn_renderer,
    [0.3, 0.35, 0.4],
    ds=0.2,
    num_steps=500,
    fn_style_by_num_traj=lambda n: ('-r', '-g', '-b')[n])
t1 = time.time()
demo1_schwarzschild_trajectories(
    fn_renderer,
    [0.3, 0.35, 0.4],
    ds=0.2,
    num_steps=500,
    fn_style_by_num_traj=lambda n: ('-r', '-g', '-b')[n])
t2 = time.time()
demo1_schwarzschild_trajectories(
    fn_renderer,
    [0.35] * 300,
    ds=0.2,
    num_steps=500,
    fn_style_by_num_traj=lambda n: ('-r', '-g', '-b')[n % 3])
t3 = time.time()
print('Timings', t1-t0, t2-t1, t3-t2)
# With GPU acceleration, this prints something like:
# Timings 3.527270555496216 3.01137113571167 4.59316873550415
#
# So, thanks to GPU hardware, producing 300 trajectories is still about
# the same effort as producing one trajectory.
# This depends of course on the capacity of the GPU units in the system.
#
# Overall, if we compute one trajectory on GPU, we may just as well
# make the extra silicon on the GPU which would only idle during that time
# do the same calculations for other trajectories.

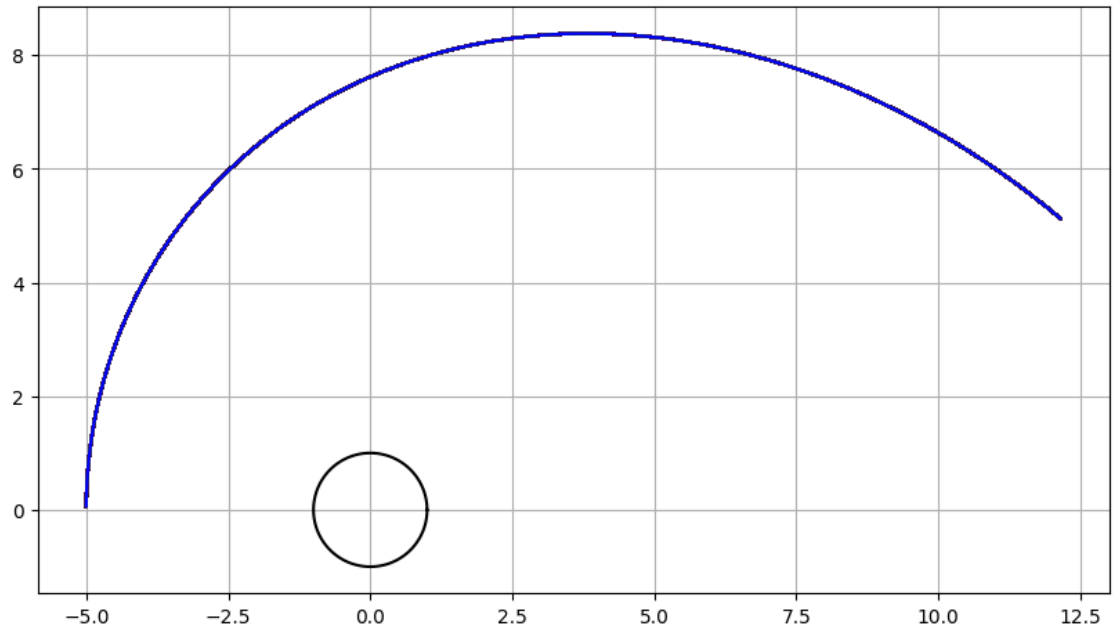
# Participant Exercise: Redo the computation with ds=0.1 and num_steps=1000
# to validate that step size is small enough to not produce
# visibly bad accuracy.
```

```
<ipython-input-22-ec64963b6829>:86: UserWarning: Matplotlib is currently using
module://matplotlib_inline.backend_inline, which is a non-GUI backend, so cannot
show the figure.
  fig.show()
```

```
Timings 3.4908783435821533 3.330188274383545 5.980652332305908
```


20
15
10
5
0
−5.0
−2.5
0.0
2.5
5.0
7.5
10.0
12.5


20
15
10
5
0
−5.0
−2.5
0.0
2.5
5.0
7.5
10.0
12.5



One thing is a bit awkward here: We are doing batch-ODE-integration with NumPy, and so end up shuffling data back and forth between GPU and CPU, once per ODE-step.

With a bit of extra cleverness, we could translate ODE-integration also to TensorFlow. This would ask for techniques such as "looping and stopping gradients" (since we are not interested in gradients here) that go beyond what we cover here.

Overall, with Colab kernel power as observed at the time this was written, we can do 300 pixels in 5 seconds, so 18k pixels in 5 minutes. This is good enough for a 128x128 image. So, let us see if we can render a crude image of the hole - even for more generic metric fields.

Here, however, it would be a shame to not exploit rotational symmetry. After all, if we merely shoot 1000 light rays at the hole, this should give us suffcient numerical information about scattering to then just rotate these sample trajectories in order to create an image.

But let us try to do this accurately. For the timelike trajectories we had earlier, we used an initial tangent-vector for ODE-integration that was not unit-normalized w.r.t. the geometry at the starting point. Since we just wanted to see "some first geodesics" for a quick plausibility-check, this did not matter much. This time, we need to do better. Why not by working out a local Lorentz frame? We will use another little helper from the author's M-Theory repository, included here to make the notebook self-contained.

[ ]:
```
def get_gramian_onb(gramian, eigenvalue_threshold=1e-7):
  """Computes orthogonalizing transform for a gramian.
  Args:
    gramian: [N, N]-array G.
    eigenvalue_threshold: Eigenvalues smaller than this
      are considered to be equal to zero.
```

```
  Returns:
    A pair of matrices (R, R_inv) such that R @ R_inv = numpy.eye(N) and
    einsum('Aa,Bb,ab->AB', R, R, gramian) is diagonal with entries
    in (0, 1, -1).
    Example: If gramian=numpy.array([[100.0, 0.1], [0.1, 1.0]])
    then R.round(6) == numpy.array([[0.00101, -1.00005], [-0.1, -0.000101]])
    and R[0, :] as well as R[1, :] are orthonormal unit vectors w.r.t.
    the scalar product given by the gramian.
  """
  gramian = numpy.asarray(gramian)
  sprods_eigvals, sprods_eigvecsT = numpy.linalg.eigh(gramian)
  abs_evs = abs(sprods_eigvals)
  onbi_scaling = numpy.where(abs_evs <= eigenvalue_threshold,
                             1.0,
                             numpy.sqrt(abs_evs))
  onbi = numpy.einsum('WZ,Z->WZ',
                      sprods_eigvecsT, onbi_scaling)
  onb = numpy.einsum('WZ,Z->WZ',
                     sprods_eigvecsT, 1.0 / onbi_scaling).T
  assert numpy.allclose(onb @ onbi, numpy.eye(onb.shape[0]))
  return onb, onbi
```

[ ]:
```
# This is the "TensorFlow-powered Virtual Event Horizon Telescope (TVEHT) :-)"

TVEHT_CAMERA_POS = [0, -20, 0, 0]

# Terminology: {name}_DmDn means the tensor has a
# (D)own-index m and (D)own-index n. 'U' is for an up-index.
CAMERA_G_DmDn = tfb_schwarzschild_metric_handcoded(
    # batching and unbatching
    tff64(TVEHT_CAMERA_POS)[tf.newaxis, ...]).numpy()[0]
tprint(CAMERA_G_DmDn, d=6, name='Camera G_AB')
# a,b,... = Lorentz index, m,n,... = Manifold-coordinates index.
CAMERA_DaUm, CAMERA_DmUa = get_gramian_onb(CAMERA_G_DmDn)
print('=== Lorentz frame metric ===\n',
      (CAMERA_DaUm @ CAMERA_G_DmDn @ CAMERA_DaUm.T).round(6))
print('=== Lorentz frame transform ===\n', CAMERA_DaUm)

# In general, the local Lorentz frame may be rotated/boosted in a weird way.
# Let us convert three relevant directions in the manifold-frame
# to the Lorentz frame: timelike, towards-the-hole, perpendicular.
# LF == Lorentz-Frame
#
# G is: G_AB, CAMERA_LM is: CAMERA_L^M, CAMERA_ML is: CAMERA_M^L

# Not-normalized Lorentz-frame vectors which in manifold-frame are:
# timelike, towards-hole, perpendicular.
```

```
LF_T0 = CAMERA_DmUa.T.dot([1, 0, 0, 0])
LF_RX0 = CAMERA_DmUa.T.dot([0, 1, 0, 0])
LF_RY0 = CAMERA_DmUa.T.dot([0, 0, 1, 0])

# We want to unit-normalize these.
ETA = numpy.diag([-1, 1, 1, 1])
LF_T = LF_T0 / (-numpy.einsum('ab,a,b->', ETA, LF_T0, LF_T0))**.5
LF_RX = LF_RX0 / numpy.einsum('ab,a,b->', ETA, LF_RX0, LF_RX0)**.5
LF_RY = LF_RY0 / numpy.einsum('ab,a,b->', ETA, LF_RY0, LF_RY0)**.5


def get_initial_lightlike_geodesic_tangent(*, x=1, y=0):
  v_xy = x * LF_RX + y * LF_RY
  norm = numpy.linalg.norm(v_xy)
  tangent_Ua = (numpy.array([norm, 0, 0, 0]) + v_xy) / norm
  tangent_Um = CAMERA_DaUm.T.dot(tangent_Ua)
  return (tangent_Ua, tangent_Um)


if 'DEBUG':
 v0_Ua, v0_Um = get_initial_lightlike_geodesic_tangent(x=1, y=0.01)
 print('\n\n=== mostly-towards-hole slightly-y-tilted light ray ===')
 print('ddd_v0_Ua:', v0_Ua.round(10).tolist())
 print('ddd_v0_Um:', v0_Um.round(10).tolist())
 print('Manifold-frame ds^2:',
       numpy.einsum('ab,a,b->', CAMERA_G_DmDn, v0_Um, v0_Um))
 v0_Um_time = numpy.pad(v0_Um[:1], [(0, 3)])
 print('Manifold-frame dt^2:',
       numpy.einsum('ab,a,b->', CAMERA_G_DmDn, v0_Um_time, v0_Um_time))
```

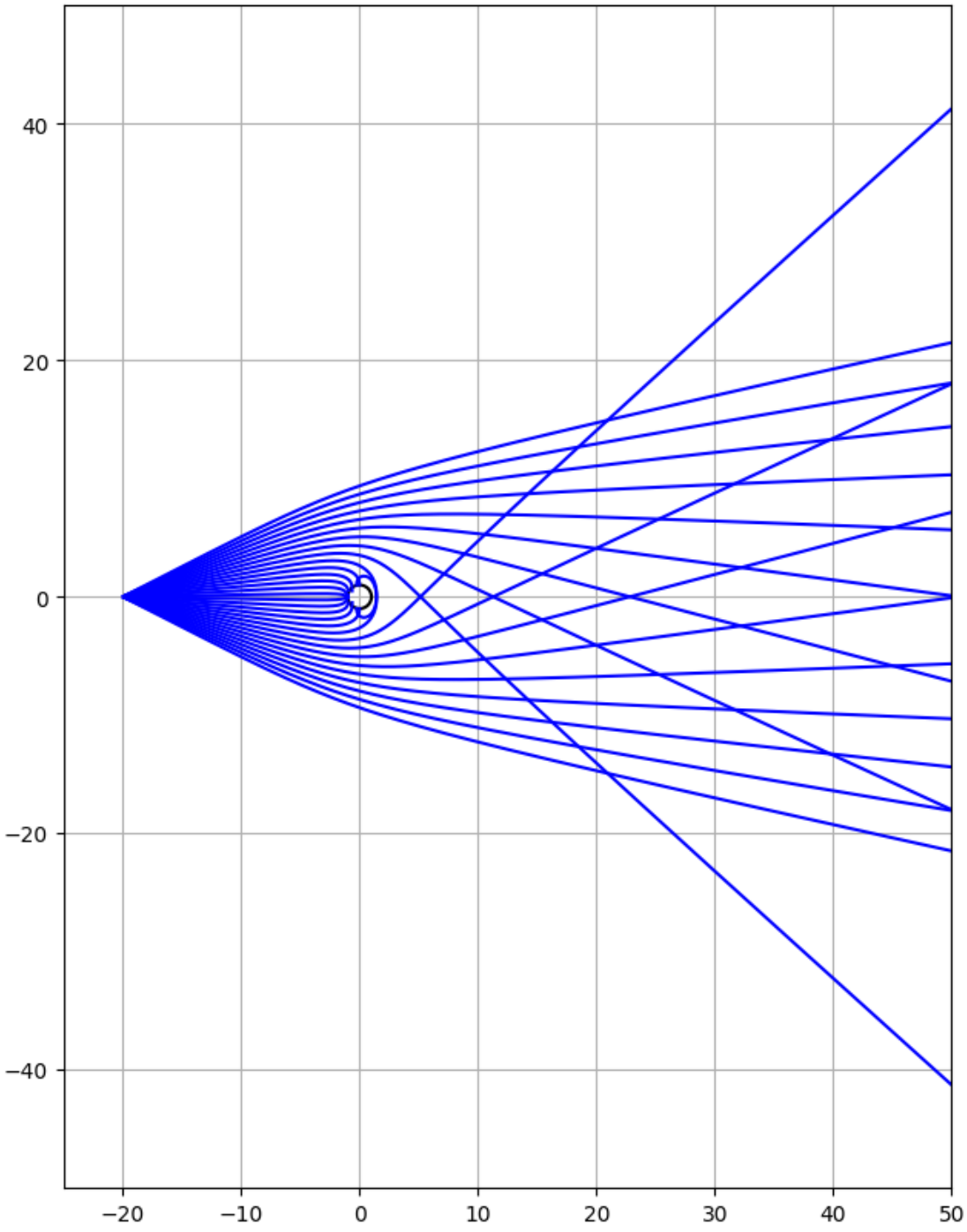
```
=== Camera G_AB, shape=(4, 4), 4 / 16 non-small entries ===
0 0: -0.95    | 1 1: 1.052632 | 2 2: 1.0      | 3 3: 1.0      |               |
|
=== Lorentz frame metric ===
 [[-1.  0.  0.  0.]
 [ 0.  1.  0.  0.]
 [ 0.  0.  1.  0.]
 [ 0.  0.  0.  1.]]
=== Lorentz frame transform ===
 [[1.02597835 0.         0.         0.        ]
 [0.         0.         1.         0.        ]
 [0.         0.         0.         1.        ]
 [0.         0.97467943 0.         0.        ]]


=== mostly-towards-hole slightly-y-tilted light ray ===
ddd_v0_Ua: [1.0, 0.0099995, 0.0, 0.9999500037]
ddd_v0_Um: [1.0259783521, 0.9746307042, 0.0099995, 0.0]
```

```
Manifold-frame ds^2: 3.164921666756748e-16
Manifold-frame dt^2: -1.0000000000000002
```

[ ]:
```python
# Let us use this to render some geodesics...

light_ys = numpy.linspace(-0.5, 0.5, 16)


def get_rays(light_ys, ds=0.1, num_steps=1000):
  trajectories = fn_renderer(
      [TVEHT_CAMERA_POS for _ in light_ys],
      [get_initial_lightlike_geodesic_tangent(y=y, x=1)[1] for y in light_ys],
      ds=ds)
  xs_vs_ss = list(itertools.islice(trajectories, num_steps))
  # We want the result to be indexed in the form:
  # result[num_trajectory, num_step, xv_selector, xyzt_coord]
  the_xs = numpy.stack([xs for xs, vs, ss in xs_vs_ss], axis=1)
  the_vs = numpy.stack([vs for xs, vs, ss in xs_vs_ss], axis=1)
  return numpy.stack([the_xs, the_vs], axis=2)

demo_rays = get_rays(numpy.linspace(-0.5, 0.5, 30))

fig = pyplot.figure(figsize=(10, 10))
ax = fig.gca()
ax.grid()
ax.set_aspect('equal')
alphas = numpy.linspace(0, 2*numpy.pi, 201)
ax.plot(numpy.cos(alphas), numpy.sin(alphas), '-k')
ax.set_xlim(-25, 50)
ax.set_ylim(-50, 50)
for geodesic in demo_rays:
  ax.plot(geodesic[:, 0, 1], geodesic[:, 0, 2], '-b')
fig.show()
```

```
<ipython-input-25-5241dd76335d>:30: UserWarning: Matplotlib is currently using
module://matplotlib_inline.backend_inline, which is a non-GUI backend, so cannot
show the figure.
  fig.show()
```



The above figure suggests that camera-placement and field-of-view choices are somewhat reasonable. Let us hence proceed to rendering.

We will need a collection of trajectories more finely spaced than what we had before. The y-range we used will correspond to the largest-angle-from-center pixels of the image, which will sit at the corners. Let us obtain 1024 trajectories.

[ ]:
```
# This takes about 3 minutes of computation
# (at the time this course material was written).
NUM_RAYS_FOR_RENDERING = 1024
SCATTERING_B_MAX = 0.7  # Somewhat larger field-of-view than above.
rendering_ys = numpy.linspace(0, SCATTERING_B_MAX, NUM_RAYS_FOR_RENDERING)
t0 = time.time()
# One-sided only, y increasing from 0 out.
#
rendering_rays = get_rays(rendering_ys, ds=0.02, num_steps=10_000)
t1 = time.time()
print(f'Obtained {NUM_RAYS_FOR_RENDERING} geodesics in {t1-t0:.3f} sec.')
```

```
Obtained 1024 geodesics in 129.186 sec.
```

We are only interested in the direction of the final velocity-vector, if this is far-out. Let us imagine there is a checkerboard at large distance behind our black hole. We are interested in light rays that passed well past the hole, i.e. where x-final is "large" (say, > 50), and the direction these rays are then traveling.

We want to associate the following colors to pixels: * Black for rays that have NaN-values in the final tangent-vector. These have hit the event horizon. * Dark Gray for "stray rays" that end up not with $x > 50, v_x > 0$. * A lighter or darker color of gray, depending on which checkerboard-cell the v_x-scaled-to-1 final velocity-vector hits.

We have our set of geodesics, but will interpolate in between them and rotate them around, exploiting some of the symmetry of the Schwarzschild solution.

[ ]:
```
import scipy.interpolate

# First, let us play a bit with the rendering-geodesics we have, and
# check a few end-states.
for index in [1000, 500, 400, 390, 380, 375, 370]:
  print(f'N={index}:\n{rendering_rays[index, -1]}')

fn_final_xv = scipy.interpolate.interp1d(
    rendering_ys,
    rendering_rays[:, -1, :, :],
    axis=0,
    # If we used higher-order interpolation, this would increase
    # the 'blast radius' of Not-a-Number values.
    kind='linear')
```

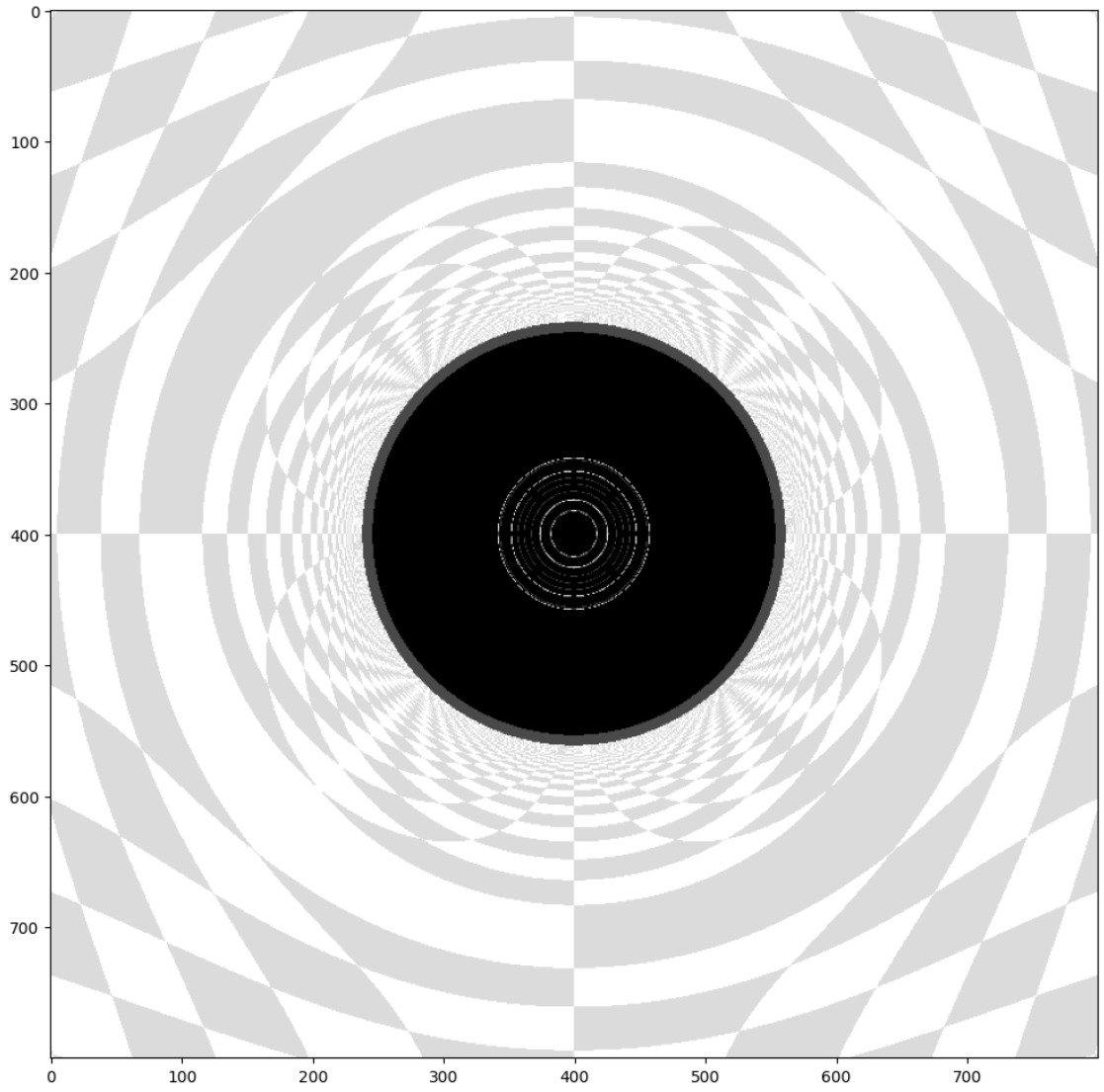
```
N=1000:
[[199.87556006 155.44916702  85.65445942   0.        ]
 [  0.98020215   0.88676353   0.40568178   0.        ]]
N=500:
[[ 2.01676029e+02  1.73408854e+02 -8.62193292e+00  0.00000000e+00]
 [ 9.80325722e-01  9.70715958e-01 -8.80635927e-02  0.00000000e+00]]
N=400:
[[202.62184955 164.58482327 -53.49140948   0.        ]
```

```
 [  0.98034422   0.91856822  -0.32687894   0.        ]]
N=390:
[[202.75571734 162.4114513  -59.67900416   0.        ]
 [  0.98034522   0.90618773  -0.35996201   0.        ]]
N=380:
[[202.90155791 159.77697873 -66.35801079   0.        ]
 [  0.98034588   0.89124621  -0.39570856   0.        ]]
N=375:
[[202.97966947 158.25089089 -69.90565055   0.        ]
 [  0.98034607   0.88261491  -0.41471078   0.        ]]
N=370:
[[203.06169927 156.56141582 -73.60693908   0.        ]
 [  0.98034613   0.87307526  -0.434547     0.        ]]
```

[ ]:
```
NUM_PIXELS = 800
R_MAX = SCATTERING_B_MAX * 0.999
X_MAX = R_MAX / 2**.5

xcoords, ycoords = numpy.meshgrid(numpy.linspace(-X_MAX, X_MAX, NUM_PIXELS),
                                  numpy.linspace(-X_MAX, X_MAX, NUM_PIXELS))

img_bs = numpy.hypot(xcoords, ycoords)
# For checkerboard-cell checking, we need the x- and y-component of a radial
# unit vector from the image-center to the given image-pixel.
img_uxs = xcoords / img_bs
img_uys = ycoords / img_bs
img_final_xvs = fn_final_xv(img_bs)
img_scattering_tan_angle = img_final_xvs[:, :, 1, 2] / img_final_xvs[:, :, 1, 1]


img_blackhole = numpy.where(
    numpy.isfinite(img_scattering_tan_angle),
    numpy.where(
        # Final x-coordinate is >50, i.e. we passed the hole.
        (img_final_xvs[:, :, 0, 1] > 50) &
        # Final x-velocity is >0, i.e. we are looking
        # "towards the sky in front of us".
        (img_final_xvs[:, :, 1, 1] > 0),
        # Case: We are looking at the checkerboard.
        # We have to work out the cell.
        0.75 + (
            numpy.floor(15 * img_scattering_tan_angle * img_uxs).astype(numpy.
 ↪int32) +
            numpy.floor(15 * img_scattering_tan_angle * img_uys).astype(numpy.
 ↪int32))
        % 2 * 0.125,
        # Case: "Stray ray, such as backscattered or still-orbiting".
```

```
        0.25
    ),
    # Case: NaN angle
    0)

# print(img_final_xvs[:5, :5, 0, 1:].round(3))
fig = pyplot.figure(figsize=(12, 12))
ax = fig.gca()
ax.imshow(img_blackhole, cmap='gray')
fig.show()
```

```
<ipython-input-29-2fb71661f656>:41: UserWarning: Matplotlib is currently using
module://matplotlib_inline.backend_inline, which is a non-GUI backend, so cannot
show the figure.
  fig.show()
```



What have we accomplished? We now have a way to do raytracing starting from an arbitrary geometry - which we might have given not even as a formula, but simply as code - in the form of a TensorFlow function that implements the spacetime-dependent ds^2.

Here, we actually only demonstrated that TensorFlow can derive a Schwarzschild-metric-function from a ds^2-function, but then (for better performance) used our hand-written Schwarzschild-metric-function afterwards (for performance reasons - since that had a simpler graph). We could of course simply swap it out.

Participant Exercise: Check the code and its output for correctness.

(The course author quickly cobbled it all together, there still might be bugs in V1 of this course! One thing that looks suspicious is that the ODE-integrator might have "jumped over the hole" for some angles, and this might potentially explain the inner rings, which likely are artefacts. Another issue is that RK4 is not really that well-suited for this kind of ODE-integration.)

19 Advanced Topics (. . . tying up some ends. . .)

This unit discusses some useful-to-know concepts that round off the course.

19.1 Extending Keras

So far, we have encountered TensorFlow in two different guises:

- As a RM-AD tool for GPU-enhanced computation of fast good-quality gradients (such as: for optimization).
- As a ML toolkit to quickly wire up some DNN architectures.

What we have not seen is how these things fit together, specifically: how can we wrap up some of our own (possibly quite sophisticated) function designs as a Keras layer?

Let us explore this by means of an example. We start with a simple convolutional architecture for CIFAR-10 one-out-of-10 image classification.

[ ]:
```
import numpy
import tensorflow as tf
import tensorflow_datasets as tfds

# Let us load the CIFAR-10 dataset rather than MNIST
# (10 classes, 32x32 pixels, RGB).
# Details: http://www.cs.toronto.edu/~kriz/cifar.html

(ds_train_raw, ds_validation_raw, ds_test_raw), ds_info = tfds.load(
    'cifar10',
    split=['train[:75%]', 'train[75%:]', 'test'],
    shuffle_files=True,
```

```
    as_supervised=True,
    with_info=True)

def normalize_image(image, label):
  """Normalizes images."""
  return tf.cast(image, tf.float32) / 255., label


ds_train = (
    ds_train_raw
    # Details: see https://www.tensorflow.org/guide/data_performance
    .map(normalize_image, num_parallel_calls=tf.data.AUTOTUNE)
    .cache()
    .shuffle(ds_info.splits['train'].num_examples)
    .batch(32)
    .prefetch(tf.data.AUTOTUNE))

ds_validation = (
    ds_validation_raw
    .map(normalize_image, num_parallel_calls=tf.data.AUTOTUNE)
    .batch(32)
    .cache()
    .prefetch(tf.data.AUTOTUNE))

ds_test = (
    ds_test_raw
    .map(normalize_image, num_parallel_calls=tf.data.AUTOTUNE)
    .batch(32)
    .cache()
    .prefetch(tf.data.AUTOTUNE))
```

[ ]:
```
### The model.

cnn_model = tf.keras.models.Sequential(
    [
        tf.keras.layers.Input(shape=(32, 32, 3)),
        tf.keras.layers.Conv2D(32, kernel_size=(3, 3), activation="relu"),
        tf.keras.layers.MaxPooling2D(pool_size=(2, 2)),
        tf.keras.layers.Conv2D(64, kernel_size=(3, 3), activation="relu"),
        tf.keras.layers.MaxPooling2D(pool_size=(2, 2)),
        tf.keras.layers.Flatten(),
        tf.keras.layers.Dropout(0.5),
        tf.keras.layers.Dense(50, activation='relu'),
        tf.keras.layers.Dense(10),
    ]
)
```

```
cnn_model.compile(
    optimizer=tf.keras.optimizers.Adam(1e-4),
    loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
    metrics=[tf.keras.metrics.SparseCategoricalAccuracy()],
)

cnn_model.fit(
    ds_train,
    epochs=25,
    validation_data=ds_validation)
```

As we have seen, modern DNN architectures tend to develop calibration problems more readily than earlier architectures.

One plausible hypothesis here might be that earlier architectures used rather simple features weakly indicative of the target class, which then also can be regarded as reasonably independent - and we rarely would encounter a situation where we accumulated "really strong evidence" where the "independency of (Bayesian) votes" would have been violated badly. With DNNs, we can have a situation where hierarchical extraction of features might lead to "tell-tale sign" type of evidence which (by optimization) gets attributed some large logit-value(s), but is not independent of other such evidence, which we do however implicitly assume by summing.

So, one could have the idea that, perhaps, large accumulated evidence (as input to softmax) should generally be distrusted, and the model is driven to use large evidence-contributions in some cases since, overall, this still improves predictions. So, how about introducing a layer that uniformly "punishes" large evidence before we feed it into softmax? Rather than trying to come up with a principled way to do this, let us here try an ad-hoc approach and attenuate all "large evidence" the same way with a nonlinear function that involves two parameters that are learnable - one determining a "length scale" below which behavior is mostly linear, and one for overall rescaling. We will go with the function `f(x) = A * asinh(B * x)`, with learnable `A` and `B`.

Also, for the sake of this example, we will ignore the question how to perhaps achieve this via creative use of some existing Keras layers (specifically, a "1D" convolution across a hidden feature-vector with an 1×1 kernel and nonlinear activation). Rather, we want to implement our own layer.

For this, we need to do a little bit of OO programming, which we so far did not discuss in this course - implementing a subclass of the `tf.keras.layers.Layer` class. This will have a tiny amount of state - just two parameters - but show all the relevant bits and pieces.

Also mostly for illustration purposes, we give our layer an extra tuning parameter which allows selecting a nonlinearity, with (here) two possible choices.

We mostly follow the Google/Alphabet Python style guide for this code.

[ ]:
```
# Introducing a new type of Keras layer.
# Base class documentation:
#   https://www.tensorflow.org/api_docs/python/tf/keras/layers/Layer
```

```
# Here, we are moving the computational body to separate @tf.function-s.
# For this particular example, the body is so simple that this makes
# little sense - we could just have done the computation in-place.
#
# In general, it is useful to have nontrivial transformations which we
# put into Keras layers also available directly as @tf.function functions
# that do not depend on Keras. A typical design would then use one module
# with such @tf.function definitions that can be used independently, plus
# a Keras wrapper that defines a layer using these functions on top of that.

@tf.function
def squash_evidence_asinh(t_evidence, t_param_a, t_param_b):
  """Squashes evidence `E` to `a * asinh(b * E)`."""
  return t_param_a * tf.math.asinh(t_param_b * t_evidence)


@tf.function
def squash_evidence_atan(t_evidence, t_param_a, t_param_b):
  """Squashes evidence `E` to `a * atan(b * E)`."""
  return t_param_a * tf.math.atan(t_param_b * t_evidence)


class EvidenceTweakingLayer(tf.keras.layers.Layer):
  """Layer for uniform nonlinear total-evidence-adjustment.

  Based on the hypothesis that seeing large accumulated evidence may generally
  have violated the Bayesian "independence" assumption beyond what we would
  be comfortable with.
  """

  # Class attributes.
  _NONLINEARITY_BY_TAG = dict(asinh=squash_evidence_asinh,
                              atan=squash_evidence_atan)

  def __init__(self, *, nonlinearity='asinh', **kwargs):
    """Initializes the instance."""
    # Forward parent-class keyword args to parent-class __init__, so that
    # `name=...` etc. args work as for a generic Keras `Layer`.
    super().__init__(**kwargs)
    nonlinearity_func = self._NONLINEARITY_BY_TAG.get(nonlinearity)
    if nonlinearity_func is None:
      raise ValueError(f'Unknown nonlinearity: {nonlinearity!r} - '
                       f'known: {set(self._NONLINEARITY_BY_TAG)!r}')
    # We need to store the __init__() parameters for .get_config(), so that
    # serialization and deserialization of the layer works.
    config = dict(nonlinearity=nonlinearity)
    config.update(**kwargs)
```

```
    self._config = config
    self._nonlinearity_func = nonlinearity_func
    # It is generally good practice to make sure that inspection of the
    # __init__ method's body gives clarity about all (public and private)
    # instance attributes.
    self._param_ab = None

  def build(self, input_shape):
    """Sets up layer-state."""
    del input_shape  # Unused by this layer.
    self._param_ab = self.add_weight(shape=(2,),
                                     initializer='random_normal')

  def call(self, inputs):
    """Evaluates the layer."""
    # Note that `inputs` may in general have batch-indices. The code in this
    # method needs to be able to handle data in such a form.
    # Here, the calculation is rather simple.
    return self._nonlinearity_func(inputs, self._param_ab[0], self._param_ab[1])

  def get_config(self):
    """Returns the layer's configuration."""
    # This method is important to ensure that saving and loading a layer
    # (such as: as part of a trained model) can re-create the given layer
    # in the expected form. Here, we need this since we have
    # instantiation-time tweaking parameters.
    #
    # The default .from_config() classmethod works for our use case,
    # since all our config is JSON-serializable.
    return self._config
```

Let us see if adding our new layer improves things.

[ ]:
```
cnn_model_tweaked = tf.keras.models.Sequential(
    [
        tf.keras.layers.Input(shape=(32, 32, 3)),
        tf.keras.layers.Conv2D(32, kernel_size=(3, 3), activation="relu"),
        tf.keras.layers.MaxPooling2D(pool_size=(2, 2)),
        tf.keras.layers.Conv2D(64, kernel_size=(3, 3), activation="relu"),
        tf.keras.layers.MaxPooling2D(pool_size=(2, 2)),
        tf.keras.layers.Flatten(),
        tf.keras.layers.Dropout(0.5),
        tf.keras.layers.Dense(50, activation='relu'),
        tf.keras.layers.Dense(10),
        EvidenceTweakingLayer(nonlinearity='asinh'),
    ]
)
```

```
cnn_model_tweaked.compile(
    optimizer=tf.keras.optimizers.Adam(1e-4),
    loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
    metrics=[tf.keras.metrics.SparseCategoricalAccuracy()],
)

cnn_model_tweaked.fit(
    ds_train,
    epochs=25,
    validation_data=ds_validation)
```

Let us try saving and loading our model.

[ ]:
```
cnn_model_tweaked.save('cnn_model_tweaked.h5')
reloaded_cnn_model_tweaked = tf.keras.models.load_model(
    'cnn_model_tweaked.h5',
    custom_objects={'EvidenceTweakingLayer': EvidenceTweakingLayer})


print('Test set accuracy (orig): '
      f'{cnn_model_tweaked.evaluate(ds_test)[1] * 100:.2f}%')
print('Test set accuracy (reloaded): '
      f'{reloaded_cnn_model_tweaked.evaluate(ds_test)[1] * 100:.2f}%')
```

```
313/313 [==============================] - 2s 7ms/step - loss: 1.1486 -
sparse_categorical_accuracy: 0.6017
Test set accuracy (orig): 60.17%
313/313 [==============================] - 1s 3ms/step - loss: 1.1486 -
sparse_categorical_accuracy: 0.6017
Test set accuracy (reloaded): 60.17%
```

Here, the classifier performance impact of this little idea was unconvincing - but this is often how things turn out when exploring ideas. Two things matter:

1. Having developed some useful intuition about what might work (via experimenting).
2. Being able to quickly try out ideas with little effort.

19.2 TensorFlow and JAX

TensorFlow is Google's ML flagship library. This however does not mean that everybody at Google would do ML exclusively with TensorFlow.

One particularly interesting "not officially supported, currently-under-research" tool which Google also open sourced is JAX.

Tony Hoare is claimed to have said that "inside every large program is a small program struggling to get out", and JAX can be thought of as being such a "small program": On its github page, it describes itself as "Autograd and XLA" (where "XLA" is the "accelerated linear algebra" also powering TensorFlow). JAX looks a lot like "numpy with automatic differentiation capabilities".

The Google colab kernel has JAX pre-installed, so let us explore it a bit.

[ ]:
```
import jax
from jax import numpy as jnp


def box_volume(sides):
  return jnp.prod(sides)

print(box_volume(jnp.array([2, 3, 4, 5], dtype=jnp.float32)))

grad_vol = jax.jit(jax.grad(box_volume))

print('Grad(volume) at [2, 4, 8]:',
      grad_vol(jnp.array([2, 4, 8], dtype=jnp.float32)))
```

```
120.0
Grad(volume) at [2, 4, 8]: [32. 16.  8.]
```

While this certainly looks useful, please note the JAX documentation (at the time of this writing) says:

```
This is a research project, not an official Google product.
Expect bugs and sharp edges.
Please help by trying it out, reporting bugs, and letting us know what you think!
```

20 Recap

1. We looked into the structure of the Python language.
   - This was a lot of material.
   - Further on, we used nested functions a lot, used numpy vectorization left and right, and having a good model of evaluation semantics was relevant for understanding how `@tf.function` sees code differently.
2. We looked into the basic principles underlying fast gradients - and their use in numerical optimization.
   - This is the basis for many ML and ML-related frameworks (other than TensorFlow and JAX).
   - This also explains the design limitations one would face when trying to implement similar such ML-supporting infrastructure on top of a framework such as Matlab, Octave, Mathematica, Maple, etc.
   - ML frameworks have their limitations, such as having little reason to support better-than-float64 numerical accuracy. So, when we cannot use them, it is good to see how we can do this on our own. Also, a solid understanding of sensitivity backpropagation opens up new perspectives on other ideas, including Hamiltonian mechanics.
   - We have seen that "solving non-malicious 1000-parameter optimization problems numerically" is generally "easy" these days.

3. We formulated a first ML problem ("digit-8-recognizer") purely as a high-dimensional optimization problem, and from there went on to ML.
    - Mental model of supervised ML: "infinite-examples limit of k-Nearest-Neighbors classifiers".
    - Major concepts we encountered: "Estimating gradients on batches" ("stochastic gradient descent"), some "standard constructions" (loss functions, softmax, embeddings), generalization-performance-enhancing tricks (early stopping, L2 regularization, dropout).
    - We looked into (often entropy-based) explanations of the design of some standard constructions.
    - We discussed what generally happens if inference is done on examples drawn on a different distribution than the one used to get the training set.
    - We briefly discussed "vanishing gradients" and how information propagation in a DNN can be seen as a percolation problem (via MFA).
    - We observed that TensorFlow can indeed make our life much easier as long as we are within the confines of what we can build with commonly used "lego bricks".
4. We took a deeper look at TensorFlow
    - We explored a bit of the inner mechanics.
    - We saw how to use it to conveniently formulate high dimensional optimization problems in a device-agnostic way (and do physics with that).
    - We saw how to wrap up "unusual" computations to make them available as Keras layers.

[Author's note: Overall, I may occasionally have talked a bit of nonsense, but this was less than 10% of the time, and physics course material rarely is right more than 90% of the time anyhow. This was the 1st iteration of teaching this material. We might want to refine this for future iterations.]

20.1 Addendum: Connecting Mathematica and TensorFlow

Given the large popularity of symbolic algebra packages such as in particular Mathematica in theoretical physics, it makes sense to address some obvious questions around ML and Mathematica.

As we have seen, the changes required to make a major programming language support backpropagation go rather deep, and with both TensorFlow and JAX, there are still some sharp edges that are related to the attempt to blend object-language (for tensor-arithmetic computations - in TensorFlow1, these enter the scene very explicitly as computational graphs) and meta-language (for setting up and manipulating object-language entities - so, Python). Design-wise, it is perhaps debateable if this is the best possible design approach. While a clearer separation between these roles would be conceptually more elegant, there is also a large human element here. In any case, it is clear that bringing backpropagation to any major language is most feasible if the language is designed with simplicity and minimality in mind (such as the Lisp dialect Scheme, where this has indeed been accomplished), or started out with the design idea of properly supporting backpropagation. Retrofitting this into some existent design of a large language is hard. Also, compiling linear algebra to machine code that can execute fast on a range of very different hardware architectures (CPUs and also GPUs) requires major effort to build.

As such, it is not clear when - or even if at all - data manipulation packages that are popular with physicists will implement advanced numerical tensor backpropagation capabilities roughly on par with TensorFlow or JAX. Clearly, major symbolic algebra packages these days do support some form of ML - but typically not in the general-purpose way that would allow us to design

and shape rather freely what our models look like. Still, one clearly would want to be able to *utilize* what such advanced capabilities have to offer in a setting where one would not want the ML library to dictate the programming language to use. This raises the question: if we might not get backpropagation with fast compiled linear algebra in the near future in popular symbolic algebra packages, is there perhaps at least a way to utilize such functionality in such a way that we can still use these packages as we are used to, but have them delegate parts of a problem to some other code behind-the-scenes? So, to the user of a Mathematica notebook, it looks as if there simply were a few functions to perform specialized data analysis, but behind the scenes, mostly invisible to the user, these functions exchange data with some other component running TensorFlow code.

This is indeed feasible - and we will look in detail into one concrete possible realization here. Before we do that, we should briefly ponder the solution landscape. There are efforts to introduce file format standards for serialized computation graphs, and modern versions of Mathematica have experimental support for loading and then running inference with models saved in the ONNX format - we produced such a serialized model earlier for classifying MNIST digits. Since this file format is rather new and new versions are currently introduced in short succession that expand the set of supported basic numerical operators, it is quite possible that the dust has to settle a bit first before this file format can be used widely without hassle. If this is available, this might be the best option.

More generally, symbolic algebra packages, like just about every programming language, usually come with a Foreign Function Interface (FFI) that allows code authors to have the runtime kernel utilize other libraries - perhaps call functions from C or Fortran libraries. In principle, it might be possible to integrate TensorFlow(-Lite) into Mathematica at this level. The advantage would be very efficient data exchange between these components. However, given that both projects are large and complex and have their own unique approaches to handling some deep technical problems, it may well be that this causes major friction. Another idea might be to go for less tight coupling and have Mathematica exchange data with another process (perhaps on the same machine) that runs TensorFlow. In Mathematica, connecting to an external process either for one-off or session-based evaluation is supported via the External Interpreted Languages Interfaces. Using this - or also a more bare-bones approach such as one based on Mathematica's RunThrough command - would typically require first serializing data into some textual form, and then communicating requests and responses back-and-forth. Overall, this requires both data-conversion and interprocess communication, so is perhaps less efficient and also more brittle (with two running processes) than a more tightly integrated solution, but also - in principle - would be much easier to build.

Since setting up a ML model is major computational effort that typically then is amortized over all the queries to that model, it makes little sense to use an approach where every query would start a new process and set up the model from saved form first - we want to go with a background server that loads the model at start time and then can be queried as needed.

A general possible concern when starting processes from Mathematica, such as via `RunThrough[]`, is that these processes generally would inherit the Mathematica kernel's process environment, which may well have adjustments to e.g. `LD_LIBRARY_PATH` (on Unix systems), making launched processes by default use a very different collection of common libraries (such as for example `libz.so`) than the system-default - and this might clash with the needs libraries needed by other components in such a set-up. Here, an extra step (such as one more indirection) would likely be needed to switch over to using a less customized library environment.

One basic fallback approach that should nowadays always be available, irrespective of which symbolic algebra system (and version) one uses, is to wire up a simple web server that allows querying ML

models via HTTP requests. This is spelled out in detail in the following - and may be a useful basis for trying very similar approaches for other combinations than Mathematica and TFLite.

20.1.1 Having Mathematica use an external HTTP server to access ML capabilities

Let us look into a basic solution for having Mathematica handle ML tasks via calling a dedicated function that delegates work to a webserver, via a HTTP request. Here, we want to consider running the webserver and the Mathematica process(es) on the same computer, which we assume to be a Unix system - such as a researcher's personal workstation. We would still want to have at least some protection against unrelated users simply accessing the webserver and submitting their own requests without any form of authorization. If the computer is the researcher's private machine, no one else can log in to it, and also all programs running on that computer can be trusted, a lightweight approach would be to simply bind the webserver port only to the loopback interface - making it unreachable from the wider network. If there are other assumed-unprivileged users (i.e. users who cannot read all files or sniff network traffic), they still could in principle access the web server socket (since web services in general cannot be run over Unix domain sockets, only TCP sockets). Here, a very simple authentication mechanism that is similar to what is used by the X Window System might make sense. The idea is that every request needs to include a "secret" which can only be learned by accessing a file on the filesystem. This way, filesystem-based access control can be transmuted into access control for services offered by a server to which every user on the local machine can connect. For X11, the `MIT-MAGIC-COOKIE-1` mechanism shall be our guideline - whatever program can read the `$HOME/.XAuthority` file (or whatever the `XAUTHORITY` environment variable has been set to) and extract the secret within can access and interact with the X display of the logged in user.

We will do something very similar with our own secrets-file: We want our webserver to write a secret to an agreed-upon location in the user's home directory at start up time, and the Mathematica kernel (which needs to be started afterwards) read that secret file and communicate the secret alongside every HTTP request to our "ML Server". For actually exchanging data, we cannot use the `HTTP GET` method for multiple reasons, an important one being URL length restrictions. Instead, we want to use `HTTP POST` requests that then transport a payload. Let us implement a bare-bones server on the basis of Python's `http` module. We want this server to - at start up time - load one (or multiple) ML models and use them to process requests, differentiating models by URL. The following code-cell is not intended to be run in a colab notebook, but is a copy-pasteable complete executable basic web server implemented in Python that can use `tflite`. We want to serve the `MNIST` tflite model we trained earlier on this course. The code comments indicate some little tricks here and there that were required to make this work despite some software packages having minor issues.

[ ]:
```
#!/usr/bin/env python3.10

import ast
import base64
import contextlib
import http.server
import threading

import io
import os
```

```
import sys

# Hack: at the time of this writing, tflite was only available for
# Python3.10, which however could not use Python3.11's numpy, so I did a:
# python3.10 -m pip install numpy -t ~/TensorFlow-ML/python3.10/site-packages
# ...and we adjust the path here.
# Note: This is for demo purposes only, and an unsound technique.
# Done properly, this would use a Python "virtual environment" (venv).
sys.path.insert(
    0,
    os.path.join(os.getenv('HOME'),
                 'TensorFlow-ML/python3.10/site-packages'))

import numpy
import tflite_runtime.interpreter as tflite


class ServerError(Exception):
  """Generic ML HTTP Server error."""


# This is somewhat slow and perhaps overkill for simply-structured data.
def parse_mathematica(data):
  """Parse mathematica via sympy.parsing."""
  # These imports only do heavy lifting upon first evaluation of the body;
  # subsequent evaluations (which however we do not have here) could
  # re-use this. If we use the below "fast" alternative, this also
  # avoids the `sympy` dependency.
  import sympy
  from sympy.parsing import mathematica
  return sympy.parsing.mathematica.parse_mathematica(data)


def parse_mathematica_simple_fast(data):
  """Ad-hoc parse simple Mathematica data (fast, lightweight)."""
  return ast.literal_eval(data.translate(str.maketrans('{}', '[]', '\n\r')))


def get_mnist_tflite_predictor(model_path):
  interpreter = tflite.Interpreter(model_path=model_path)
  interpreter.allocate_tensors()
  input_details = interpreter.get_input_details()
  output_details = interpreter.get_output_details()
  interpreter_lock = threading.Lock()
  #
  def fn_predict(in_data, verbose=False):
    # Note that this is setting interpreter-state, making
```

```
    # this function non-reentrant unless we ensure that
    # interpreter.invoke() calls cannot get mangled by
    # interwoven concurrent set-up / read-out operations.
    # This would matter a lot if we were to use e.g.
    # a http.server.ThreadingHTTPServer - but let's make this
    # robust.
    try:
      interpreter_lock.acquire()
      # Here, we are extra-permissive:
      # Any numerical data matrix/vector that comes in gets zero-padded
      # to at least 28x28 elements, then trimmed to 28x28 elements,
      # then reshaped. This allows us to easily hand-feed data such as
      # {1,2,3} for debugging. No harm in trying to predict
      # from bad-size data here.
      in_array = numpy.pad(
          numpy.asarray(in_data, dtype=numpy.float32).ravel(),
          ((0, 28*28),))[:28*28].reshape(1, 28, 28)
      interpreter.set_tensor(
          input_details[0]['index'], in_array)
      interpreter.invoke()
      output_data = interpreter.get_tensor(output_details[0]['index'])
      return output_data[0, ...].tolist()
    finally:
      interpreter_lock.release()
  #
  return fn_predict


class ML_HTTPServer(http.server.HTTPServer):
  """Server subclass that has access to a ML model.

  Attributes:
    (base class attributes plus...):
    ml_fn_predict_by_urlpath: Mapping of url-path to predictor-function,
      as documented in `__init__`.
  """

  def __init__(self, server_address, request_handler_class, *,
               fn_predict_by_urlpath=(),
               token_file='.ml_server_token'):
    """Initializes the instance.

    Args:
      server_address: The server address, passed on to base class `__init__`.
      request_handler_class: The request handler class, passed on to
        base class `__init__`.
      fn_predict_by_urlpath: data which when passed to `dict()` produces
```

```
        a key-value dictionary that maps a URL-path key to a predictor-function;
        Each predictor function is expected to map a single numpy.ndarray-like
        ML-data argument to numerical output data.
      token_file: Path (implicitly-relative to web-user's $HOME env-var)
        to a file where upon webserver startup the server writes a random
        secret. POST web requests must include the secret as the 1st line of
        the payload, proving that the requestor could read the token-file.
        This loosely resembles MIT-MAGIC-COOKIE-1 X Window Authorization and
        protects against independent users on the same machine (who see the
        TCP port bound to the loopback interface) connecting to the webserver
        and issuing their own requests. Does not protect against local
        users with elevated privileges, such as root access to files or
        packet sniffing. If web requests are to be routed from a different
        machine, TLS encryption should be used additionally.
    """
    # Secret handling goes first. Overall, this is a primitive mechanism
    # that one might want to replace with something more advanced,
    # such as HMAC, but the client also needs to support this.
    secret = base64.b64encode(os.getrandom(32)).rstrip(b'=')
    token_path = os.path.join(os.getenv('HOME'), token_file)
    with open(token_path, 'wb') as h_token:
      os.fchmod(h_token.fileno(), 0o700)
      h_token.write(secret)
    # Check if writing the secret succeeded.
    with open(token_path, 'rb') as h_token_re:
      re_secret = h_token_re.read()
    if not re_secret == secret:
      raise ServerError(
          'Could not align on-filesystem secret '
          f'with internal secret - path: {token_path}')
    # Initialize base class instance. We have not yet touched instance-state.
    super().__init__(server_address, request_handler_class)
    self.ml_fn_predict_by_urlpath = dict(fn_predict_by_urlpath)
    self._secret = secret

  def check_secret(self, user_provided):
    """Checks if the user-provided string equals the secret."""
    return self._secret == user_provided


class ML_HTTPRequestHandler(http.server.BaseHTTPRequestHandler):
  """HTTP Request handler that delegates POST to specialist functions.

  Also performs secret-checking on the server.
  """

  def _send_text_plain_response(self, status_code, response):
```

```
    self.send_response(status_code)
    self.send_header('Content-Type', 'text/plain')
    self.send_header('Content-Length', str(len(response)))
    self.end_headers()
    self.wfile.write(response)

  def do_POST(self):
    req_urlpath = self.path
    content_length = int(self.headers.get('content-length', 0))
    request_data = self.rfile.read(content_length)
    try:
      # If any of these operations fail, such as due to `req_urlpath`
      # not being in the mapping, the payload not having a b'\n', etc.,
      # the violation of the implicit code-expectation raises an exception,
      # and the `except`-section just returns a `Not Implemented` response.
      user_provided_secret, payload = request_data.split(b'\n', 1)
      if not self.server.check_secret(user_provided_secret):
          # Handled right below.
          # We provide a message since we stderr-output text.
          raise ServerError('Bad secret.')
      parsed_payload = parse_mathematica_simple_fast(payload.decode('utf-8'))
      fn_predict = self.server.ml_fn_predict_by_urlpath[req_urlpath]
      prediction = fn_predict(parsed_payload)
      response = repr(prediction).translate(
          str.maketrans('[]', '{}')).encode('utf-8')
      self._send_text_plain_response(http.HTTPStatus.OK, response)
    except Exception as exn:
      print('ERROR:', repr(exn), file=sys.stderr)
      self._send_text_plain_response(http.HTTPStatus.NOT_IMPLEMENTED,
                                     b'501 - Not Implemented')


def run_server(server_address=('localhost', 8000),
               mnist_model='mnist_model.tflite'):
  fn_predict_mnist = get_mnist_tflite_predictor(mnist_model)
  httpd = ML_HTTPServer(server_address,
                        ML_HTTPRequestHandler,
                        fn_predict_by_urlpath={
                            '/mnist': fn_predict_mnist,
                            })
  httpd.serve_forever()


if __name__ == '__main__':
  run_server()
```

If we start this webserver, the following Mathematica code illustrates how to then wrap up delegation of a ML inference task to this external server.

```
(* Server URL Authentication Secret *)
mlServerURL := "http://localhost:8000/mnist";
mlServerTokenPath:=FileNameJoin[{$HomeDirectory,".ml_server_token"}];
mlServerAuthSecret = Import[mlServerTokenPath, "Text"];
mlServerAuthSecretLength := {"Secret Length", StringLength[mlServerAuthSecret]};
Print[mlServerAuthSecretLength];


(* Running external classification *)
TFClassifyMNIST[numdata_]:=Module[{body, result},
body=mlServerAuthSecret<>"\n"<>ExportString[numdata,"String"];
result=URLFetch[mlServerURL,"Method"->"POST","Body"->body];
result]


(* Demo - Example Input *)

(* For illustration: Create 28x28 input from 7x7 text. *)

as28x28[textImage7x7_]:=KroneckerProduct[
ToExpression[#/.{"#"->1.0,"."->0.0}]&/@Characters[textImage7x7],
ConstantArray[1, {4,4}]]

demoDigit := as28x28[{
".......",
"..####.",
"..#..#.",
"..####.",
"..#..#.",
"..####.",
"......."}]

(* Running Classification *)

MLResponse:=TFClassifyMNIST[demoDigit];
Print[MLResponse]
(* Produces:
{-0.12768815457820892, -3.9151949882507324,
 -1.174180269241333, -2.3871653079986572,
 0.34610337018966675, 3.6387715339660645,
 2.232551097869873, -4.550597667694092,
 4.761049270629883, 0.7428076267242432}
*)
```

This basic template can (perhaps with a bit of help from a Unix wizard) be adjusted to other, similar tasks.

References

[1] Andrew Brock, Theodore Lim, James M Ritchie, and Nick Weston. Neural photo editing with introspective adversarial networks. *arXiv preprint arXiv:1609.07093*, 2016.

[2] Thomas Fischbacher. The many vacua of gauged extended supergravities. *General Relativity and Gravitation*, 41(2):315–411, 2009.

[3] Thomas Fischbacher, Iulia M Comsa, Krzysztof Potempa, Moritz Firsching, Luca Versari, and Jyrki Alakuijala. Intelligent matrix exponentiation. *arXiv preprint arXiv:2008.03936*, 2020.

[4] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 249–256. JMLR Workshop and Conference Proceedings, 2010.

[5] Joseph Goguen and Grant Malcolm. *Algebraic semantics of imperative programs*. MIT press, 1996.

[6] Gabriel Goh, Nick Cammarata †, Chelsea Voss †, Shan Carter, Michael Petrov, Ludwig Schubert, Alec Radford, and Chris Olah. Multimodal neurons in artificial neural networks. *Distill*, 2021. https://distill.pub/2021/multimodal-neurons.

[7] Ian J Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. *arXiv preprint arXiv:1412.6572*, 2014.

[8] Paul Graham. A plan for spam. *http://paulgraham. com/spam. html*, 2002.

[9] Chuan Guo, Geoff Pleiss, Yu Sun, and Kilian Q Weinberger. On calibration of modern neural networks. In *International conference on machine learning*, pages 1321–1330. PMLR, 2017.

[10] Laurent Hascoet and Valérie Pascual. The tapenade automatic differentiation tool: principles, model, and specification. *ACM Transactions on Mathematical Software (TOMS)*, 39(3):1–43, 2013.

[11] Günter Klambauer, Thomas Unterthiner, Andreas Mayr, and Sepp Hochreiter. Self-normalizing neural networks. *Advances in neural information processing systems*, 30, 2017.

[12] Ankur Mallick, Chaitanya Dwivedi, Bhavya Kailkhura, Gauri Joshi, and T Yong-Jin Han. Can your ai differentiate cats from covid-19? sample efficient uncertainty estimation for deep learning safety. *choice*, 50(6), 2020.

[13] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *nature*, 323(6088):533–536, 1986.

[14] Samuel S Schoenholz, Justin Gilmer, Surya Ganguli, and Jascha Sohl-Dickstein. Deep information propagation. *arXiv preprint arXiv:1611.01232*, 2016.

[15] Bert Speelpenning. *Compiling fast partial derivatives of functions given by algorithms*. University of Illinois at Urbana-Champaign, 1980.

[16] Alan Turing. Intelligent machinery (1948). *B. Jack Copeland*, page 395, 2004.