LxMLS - Lab Guide

July 19, 2025

Day 0

Basic Tutorials

0.1 Today's assignment

In this class we will introduce several fundamental concepts needed further ahead. We start with an introduction to Python, the programming language we will use in the lab sessions, and to Matplotlib and Numpy, two modules for plotting and scientific computing in Python, respectively. Afterwards, we present several notions on probability theory and linear algebra. Finally, we focus on numerical optimization. The goal of this class is to give you the basic knowledge for you to understand the following lectures. We will not enter in too much detail in any of the topics.

0.2 Manually Installing the Tools in your own Computer

0.2.1 Desktops vs. Laptops

If you have decided to use one of our provided desktops, all installation procedures have been carried out. You merely need to go to the `lxmls-toolkit-student` folder inside your home directory and start working! You may go directly to section 0.3. If you wish to use your own laptop, you will need to install Python, the required Python libraries and download the LXMLS code base. It is important that you do this as soon as possible (before the school starts) to avoid unnecessary delays. Please follow the install instructions.

0.2.2 Basic Install and Troubleshooting

To install, just follow the instructions in our Github repository for the `student` version of our toolkit

- `https://github.com/LxMLS/lxmls-toolkit/tree/student#readme`.

The `student` branch contains the same code as `master` branch, with some parts deleted, which you must complete in the following exercises. The following install instructions are same as the one on GitHub, provided here for completeness.

The basic install instructions use `uv`. If you are on Linux or MacOS, install `uv` with

```
curl -LsSf https://astral.sh/uv/install.sh | sh
```

If you are on Windows, use

```
powershell -ExecutionPolicy ByPass -c "irm https://astral.sh/uv/install.ps1 | iex"
```

If you don't have Python on your system or want to install a specific version (the labs require the Python version to be between 3.9 and 3.12) you can install it using `uv`:

```
uv python install 3.10
```

You can then install all required dependencies and create a virtual environment using the one-liner:

```
uv sync --extra cpu
```

If your device has access to an Nvidia GPU, you can replace `cpu` with `cu118`, `cu124`, or `cu126` depending on the CUDA version your GPU supports. Use `nvidia-smi` or `nvcc --version` to check the CUDA version.

To activate your virtual environment with all installed dependencies on Linux or MacOS use `source .venv/bin/activate`. To do so on Windows use `.venv`

Check if your virtual environment is active by running `python --version && which python`. This should display the correct version of python with a path from within the current virtual environment.

0.2.3 Deciding on the IDE and interactive shell to use

An Integrated Development Environment (IDE) includes a text editor and various tools to debug and interpret complex code.

Important: As the labs progress you will need an IDE, or at least a good editor and knowledge of pdb/ipdb. This will not be obvious the first days since we will be seeing simpler examples.

Easy IDEs to work with Python are PyCharm and Visual Studio Code, but feel free to use the software you feel more comfortable with. PyCharm and other well known IDEs like Spyder are provided with the Anaconda installation.

Aside of an IDE, you will need an interactive command line to run commands. This is very useful to explore variables and functions and quickly debug the exercises. For the most complex exercises you will still need an IDE to modify particular segments of the provided code. As interactive command line we recommend the Jupyter notebook. This also comes installed with Anaconda and is part of the pip-installed packages. The Jupyter notebook is described in the next section. In case you run into problems or you feel uncomfortable with the Jupyter notebook you can use the simpler iPython command line.

0.2.4 Jupyter Notebook

Jupyter is a good choice for writing Python code. It is an interactive computational environment for data science and scientific computing, where you can combine code execution, rich text, mathematics, plots and rich media. The Jupyter Notebook is a web application that allows you to create and share documents, which contains live code, equations, visualizations and explanatory text. It is very popular in the areas of data cleaning and transformation, numerical simulation, statistical modeling, machine learning and so on. It supports more than 40 programming languages, including all those popular ones used in Data Science such as Python, R, and Scala. It can also produce many different types of output such as images, videos, LaTex and JavaScript. More over with its interactive widgets, you can manipulate and visualize data in real time.
The main features and advantages using the Jupyter Notebook are the following:

- In-browser editing for code, with automatic syntax highlighting, indentation, and tab completion/introspection.
- The ability to execute code from the browser, with the results of computations attached to the code which generated them.
- Displaying the result of computation using rich media representations, such as HTML, LaTeX, PNG, SVG, etc. For example, publication-quality figures rendered by the matplotlib library, can be included inline.
- In-browser editing for rich text using the Markdown markup language, which can provide commentary for the code, is not limited to plain text.
- The ability to easily include mathematical notation within markdown cells using LaTeX, and rendered natively by MathJax.

The basic commands you should know are

Esc	Enter command mode
Enter	Enter edit mode
up/down	Change between cells
Ctrl + Enter	Runs code on selected cell
Shift + Enter	Runs code on selected cell, jumps to next cell
restart button	Deletes all variables (useful for troubleshooting)

Table 1: Basic Jupyter commands

A more detailed user guide can be found here:

`http://jupyter-notebook-beginner-guide.readthedocs.io/en/latest/index.html`

0.3 Solving the Exercises

In the student branch we provide the `solve.py` script. This can be used to solve the exercises of each day, e.g.,

```
python ./solve.py linear_classifiers
```

where the solvable days are: linear_classifiers, sequence_models, structure_predictors, non-linear_classifiers, non-linear_sequence_models, reinforcement_learning. You can also undo the solving of an exercise by using

```
python ./solve.py --undo linear_classifiers
```

Note that this script just downloads the master or student versions of certain files from the GitHub repository. It needs an Internet connection. Since some exercises require you to have the exercises of the previous days completed, the monitors may ask you to use this function. **Important:** Remember to save your own version of the code, otherwise it will be overwritten!

0.4 Python

0.4.1 Python Basics

Pre-requisites

At this point you should have installed the needed packages. You need also to feel comfortable with an IDE to edit code and an interactive command line. See previous sections for the details. Your work folder will be

```
lxmls-toolkit-student
```

from there, start your interactive command line of choosing, e.g.,

```
jupyter-notebook
```

and proceed with the following sections.

Running Python code

We will start by creating and running a dummy program in Python which simply prints the "Hello World!" message to the standard output (this is usually the first program you code when learning a new programming language). There are two main ways in which you can run code in Python:

From a file – Create a file named `yourfile.py` and write your program in it, using the IDE of your choice, e.g., PyCharm:

```
print('Hello World!')
```

After saving and closing the file, you can run your code by using the run functionality in your IDE. If you wish to run from a command line instead do

```
python yourfile.py
```

This will run the program and display the message "Hello World!". After that, the control will return to the command line or IDE.

In the interactive command line – Start your preferred interactive command line, e.g., Jupyter-notebook. There, you can run Python code by simply writing it and pressing enter (ctr+enter in Jupyter).

```
 In[]: print("Hello, World!")
Out[]: Hello, World!
```

However, you can also run Python code written into a file.

```
 In[]: run ./yourfile.py
Out[]: Hello, World!
```

Keep in mind that you can easily switch between these two modes. You can quickly test commands directly in the command line and, e.g., inspect variables. Larger sections of code can be stored and run from files.

Help and Documentation

There are several ways to get help on Jupyter:

- Adding a question mark to the end of a function or variable and pressing Enter brings up associated documentation. Unfortunately, not all packages are well documented. Numpy and matplotlib are pleasant exceptions;
- `help(’if’)` gets the online documentation for the `if` keyword;
- `help()`, enters the help system.
- When at the help system, type `q` to exit.

0.4.2 Python by Example

Basic Math Operations

Python supports all basic arithmetic operations, including exponentiation. For example, the following code:

```
print(3 + 5)
print(3 - 5)
print(3 * 5)
print(3 / 5)
print(3 ** 5)
```

will produce the following output in Python 2:

```
8
-2
15
0
243
```

and the following output in Python 3:

```
8
-2
15
0.6
243
```

Important: Notice that in Python 2 division is always considered as integer division, hence the result being 0 on the example above. To force a floating point division in Python 2 you can force one of the operands to be a floating point number:

```
print(3 / 5.0)
0.6
```

For Python 3, the division is considered float point, so the operation (3 / 5) or (3 / 5.0) is always 0.6.

Also, notice that the symbol `**` is used as exponentation operator, unlike other major languages which use the symbol ˆ. In fact, the ˆ symbol has a different meaning in Python (bitwise XOR) so, in the beginning, be sure to double-check your code if it uses exponentiation and it is giving unexpected results.

Data Structures

In Python, you can create lists of items with the following syntax:

```
countries = ['Portugal','Spain','United Kingdom']
```

A string should be surrounded by either apostrophes (′) or quotes (“). You can access a list with the following:

- `len(L)`, which returns the number of items in L;
- `L[i]`, which returns the item at index i (the first item has index 0);
- `L[i:j]`, which returns a new list, containing all the items between indexes i and $j-1$, inclusively.

Exercise 0.1 *Use L[i:j] to return the countries in the Iberian Peninsula.*

Loops and Indentation

A loop allows a section of code to be repeated a certain number of times, until a stop condition is reached. For instance, when the list you are iterating over has reached its end or when a variable has reached a certain value (in this case, you should not forget to update the value of that variable inside the code of the loop). In Python you have `while` and `for` loop statements. The following two example programs output exactly the same using both statements: the even numbers from 2 to 8.

```
i = 2
while i < 10:
  print(i)
  i += 2
```

```
for i in range(2,10,2):
    print(i)
```

You can copy and run this in Jupyter. Alternatively you can write this into your `yourfile.py` file and run it. Do you notice something? It is possible that the code did not act as expected or maybe an error message popped up. This brings us to an important aspect of Python: **indentation**. Indentation is the number of blank spaces at the leftmost of each command. This is how Python differentiates between blocks of commands inside and outside of a statement, e.g., `while` or `for`. All commands within a statement have the same number of blank spaces at their leftmost. For instance, consider the following code:

```
a=1
while a <= 3:
    print(a)
    a += 1
```

and its output:

```
1
2
3
```

Exercise 0.2 *Can you then predict the output of the following code?:*

```
a=1
while a <= 3:
    print(a)
a += 1
```

Bear in mind that indentation is often the main source of errors when starting to work with Python. Try to get used to it as quickly as possible. It is also recommendable to use a text editor that can display all characters e.g. blank space, tabs, since these characters can be visually similar but are considered different by Python. One of the most common mistakes by newcomers to Python is to have their files indented with spaces on some lines and with tabs on other lines. Visually it might appear that all lines have proper indentation, but you will get an `IndentationError` message if you try it. The recommended[1] way is to use 4 spaces for each indentation level.

Control Flow

The `if` statement allows to control the flow of your program. The next program outputs a greeting that depends on the time of the day.

```
hour = 16
if hour < 12:
    print('Good morning!')
elif hour >= 12 and hour < 20:
    print('Good afternoon!')
else:
    print('Good evening!')
```

Functions

A function is a block of code that can be reused to perform a similar action. The following is a function in Python.

```
def greet(hour):
    if hour < 12:
        print('Good morning!')
    elif hour >= 12 and hour < 20:
        print('Good afternoon!')
    else:
        print('Good evening!')
```

You can write this command into Jupyter directly or write it into a file which you then run in Jupyter. Once you do this the function will be available for you to use. Call the function `greet` with different hours of the day (for example, type `greet(16)`) and see that the program will greet you accordingly.

Exercise 0.3 *Note that the previous code allows the hour to be less than 0 or more than 24. Change the code in order to indicate that the hour given as input is invalid. Your output should be something like:*

```
greet(50)
Invalid hour: it should be between 0 and 24.
greet(-5)
Invalid hour: it should be between 0 and 24.
```

[1] The PEP8 document (`www.python.org/dev/peps/pep-0008`) is the official coding style guide for the Python language.

Profiling

If you are interested in checking the performance of your program, you can use the command `%prun` in Jupyter. For example:

```
def myfunction(x):
    ...

%prun myfunction(22)
```

The output of the `%prun` command will show the following information for each function that was called during the execution of your code:

- `ncalls`: The number of times this function was called. If this function was used recursively, the output will be two numbers; the first one counts the total function calls with recursions included, the second one excludes recursive calls.
- `tottime`: Total time spent in this function, excluding the time spent in other functions called from within this function.
- `percall`: Same as `tottime`, but divided by the number of calls.
- `cumtime`: Same as `tottime`, but including the time spent in other functions called from within this function.
- `percall`: Same as `cumtime`, but divided by the number of calls.
- `filename:lineno(function)`: Tells you where this function was defined.

Debugging in Python

During the lab sessions, there will be situations in which we will use and extend modules that involve elaborated code and statements, like classes and nested functions. Although desirable, it should not be necessary for you to fully understand the whole code to carry out the exercises. It will suffice to understand the algorithm as explained in the theoretical part of the class and the local context of the part of the code where we will be working. For this to be possible is very important that you learn to use an IDE.

An alternative to IDEs, that can also be useful for quick debugging in Jupyter, is the pdb module. This will stop the execution at a given point (called break-point) to get a quick glimpse of the variable structures and to inspect the execution flow of your program. The ipdb is an improved version of pdb that has to be installed separately. It provides additional functionalities like larger context windows, variable auto complete and colors. Unfortunately ipdb has some compatibility problems with Jupyter. We therefore recommend to use ipdb only in spartan configurations such as vim+ipdb as IDE.

In the following example, we use this module to inspect the `greet` function:

```
def greet(hour):
    if hour < 12:
        print('Good morning!')
    elif hour >= 12 and hour < 20:
        print('Good afternoon!')
    else:
        import pdb; pdb.set_trace()
        print('Good evening!')
```

Load the new definition of the function by writing this code in a file or a Jupyter cell and running it. Now, if you try `greet(50)` the code execution should stop at the place where you located the break-point (that is, in the `print('Good evening!')` statement). You can now run new commands or inspect variables. For this purpose there are a number of commands you can use[2], but we provide here a short table with the most useful ones:

Getting back to our example, we can type n(ext) once to execute the line we stopped at

[2] The complete list can be found at `http://docs.python.org/library/pdb.html`

(h)elp (p)rint (p)retty(p)rint	Starts the help menu Prints a variable Prints a variable, with line break (useful for lists)
(n)ext line (s)tep c(ont(inue)) (r)eturn b(reak) n b(reak) n, condition	Jumps to next line Jumps inside of the function we stopped at Continues execution until finding breakpoint or finishing Continues execution until current function returns Sets a breakpoint in in line n Sets a conditional breakpoint in in line n

l(ist) [n], [m]
w(here)
u(p)
d(down)
	Prints 11 lines around current line. Optionally starting in line n or between lines n, m
	Shows which function called the function we are in, and upwards (stack)
	Goes one level up the stack (frame of the function that called the function we are on)
	Goes one level down the stack

blank *expression*	Repeat the last command Executes the python expression as if it was in current frame

Table 2: Basic pdb/ipdb commands, parentheses indicates abbreviation

```
pdb> n
> ./lxmls-toolkit/yourfile.py(8)greet()
      7                 import pdb; pdb.set_trace()
----> 8                 print('Good evening!')
```

Now we can inspect the variable `hour` using the p(retty)p(rint) option

```
pdb> pp hour
50
```

From here we could keep advancing with the n(ext) option or set a b(reak) point and type c(ontinue) to jump to a new position. We could also execute any python expression which is valid in the current frame (the function we stopped at). This is particularly useful to find out why code crashes, as we can try different alternatives without the need to restart the code again.

0.4.3 Exceptions

Occasionally, a syntactically correct code statement may produce an error when an attempt is made to execute it. These kind of errors are called *exceptions* in Python. For example, try executing the following:

```
10/0
```

A *ZeroDivisionError* exception was raised, and no output was returned. Exceptions can also be forced to occur by the programmer, with customized error messages [3].

```
raise ValueError("Invalid input value.")
```

Exercise 0.4 *Rewrite the code in Exercise 0.3 in order to raise a ValueError exception when the hour is less than 0 or more than 24.*

Handling of exceptions is made with the *try* statement:

[3] For a complete list of built-in exceptions, see `http://docs.python.org/3/library/exceptions.html`

```python
while True:
    try:
        x = int(input("Please enter a number: "))
        break
    except ValueError:
        print("Oops! That was no valid number. Try again...")
```

It works by first executing the *try* clause. If no exception occurs, the *except* clause is skipped; if an exception does occur, and if its type matches the exception named in the *except* keyword, the except clause is executed; otherwise, the exception is raised and execution is aborted (if it is not caught by outer *try* statements).

Extending basic Functionalities with Modules

In Python you can load new functionalities into the language by using the `import`, `from` and `as` keywords. For example, we can load the numpy module as

```python
import numpy as np
```

Then we can run the following on the Jupyter command line:

```
np.var?
np.random.normal?
```

The import will make the numpy tools available through the alias `np`. This shorter alias prevents the code from getting too long if we load lots of modules. The first command will display the help for the method `numpy.var` using the previously commented symbol `?`. Note that in order to display the help you need the full name of the function including the module name or alias. Modules have also submodules that can be accessed the same way, as shown in the second example.

Organizing your Code with your own modules

Creating you own modules is extremely simple. you can for example create the file in your work directory

`my_tools.py`

and store there the following code

```python
def my_print(input):
    print(input)
```

From Jupyter you can now import and use this tool as

```python
import my_tools
my_tools.my_print("This works!")
```

Important: When you modify a module, you need to reload the notebook page for the changes to take effect. Autoreload is set by default in the schools notebooks.
for the latter. Other ways of importing one or all the tools from a module are

```python
from my_tools import my_print   # my_print directly accesible in code
from my_tools import *          # will make all functions in my_tools accessible
```

However, this makes reloading the module more complicated. You can also store tools ind different folders. For example, if you store the previous example in the folder

```
day0_tools
```

and store inside an empty file called

```
__init__.py
```

then the following import will work

```python
import day0_tools.my_tools
```

0.4.4 Matplotlib – Plotting in Python

Matplotlib[4] is a plotting library for Python. It supports 2D and 3D plots of various forms. It can show them interactively or save them to a file (several output formats are supported).

```python
import numpy as np
import matplotlib.pyplot as plt

X = np.linspace(-4, 4, 1000)

plt.plot(X, X**2*np.cos(X**2))
plt.savefig("simple.pdf")
```

Exercise 0.5 *Try running the following on Jupyter, which will introduce you to some of the basic numeric and plotting operations.*

```python
# This will import the numpy library
# and give it the np abbreviation
import numpy as np

# This will import the plotting library
import matplotlib.pyplot as plt

# Linspace will return 1000 points,
# evenly spaced between -4 and +4
X = np.linspace(-4, 4, 1000)

# Y[i] = X[i]**2
Y = X**2

# Plot using a red line ('r')
plt.plot(X, Y, 'r')

# arange returns integers ranging from -4 to +4
# (the upper argument is excluded!)
Ints = np.arange(-4,5)

# We plot these on top of the previous plot
# using blue circles (o means a little circle)
plt.plot(Ints, Ints**2, 'bo')

# You may notice that the plot is tight around the line
# Set the display limits to see better
plt.xlim(-4.5,4.5)
plt.ylim(-1,17)
plt.show()
```

[4] http://matplotlib.org/

0.4.5 Numpy – Scientific Computing with Python

Numpy[5] is a library for scientific computing with Python.

Multidimensional Arrays

The main object of numpy is the multidimensional array. A multidimensional array is a table with all elements of the same type and can have several dimensions. Numpy provides various functions to access and manipulate multidimensional arrays. In one dimensional arrays, you can index, slice, and iterate as you can with lists. In a two dimensional array M, you can perform these operations along several dimensions.

- M[i,j], to access the item in the i^{th} row and j^{th} column;
- M[i:j,:], to get the all the rows between the i^{th} and $j-1^{th}$;
- M[:,i], to get the i^{th} column of M.

Again, as it happened with the lists, the first item of every column and every row has index 0.

```
import numpy as np
A = np.array([
    [1,2,3],
    [2,3,4],
    [4,5,6]])

A[0,:] # This is [1,2,3]
A[0] # This is [1,2,3] as well

A[:,0] # this is [1,2,4]

A[1:,0] # This is [ 2, 4 ]. Why?
        # Because it is the same as A[1:n,0] where n is the size of the array.
```

Mathematical Operations

There are many helpful functions in numpy. For basic mathematical operations, we have `np.log`, `np.exp`, `np.cos`,...with the expected meaning. These operate both on single arguments and on arrays (where they will behave element wise).

```
import matplotlib.pyplot as plt
import numpy as np

X = np.linspace(0, 4 * np.pi, 1000)
C = np.cos(X)
S = np.sin(X)

plt.plot(X, C)
plt.plot(X, S)
```

Other functions take a whole array and compute a single value from it. For example, `np.sum`, `np.mean`,... These are available as both free functions and as methods on arrays.

```
import numpy as np

A = np.arange(100)

# These two lines do exactly the same thing
print(np.mean(A))
print(A.mean())
```

[5] http://www.numpy.org/

```
C = np.cos(A)
print(C.ptp())
```

Exercise 0.6 *Run the above example and lookup the* `ptp` *function/method (use the* `?` *functionality in Jupyter).*

Exercise 0.7 *Consider the following approximation to compute an integral*

$$\int_0^1 f(x)dx \approx \sum_{i=0}^{999} \frac{f(i/1000)}{1000}.$$

Use numpy to implement this for $f(x) = x^2$. You should not need to use any loops. Note that integer division in Python 2.x returns the floor division (use floats – e.g. 5.0/2.0 – to obtain rationals). The exact value is 1/3. How close is the approximation?

0.5 Essential Linear Algebra

Linear Algebra provides a compact way of representing and operating on sets of linear equations.

$$\begin{cases} 4x_1 - 5x_2 = -13 \\ -2x_1 + 3x_2 = 9 \end{cases}$$

This is a system of linear equations in 2 variables. In matrix notation we can write the system more compactly as

$$\mathbf{A}\mathbf{x} = \mathbf{b}$$

with

$$\mathbf{A} = \begin{bmatrix} 4 & -5 \\ -2 & 3 \end{bmatrix}, \mathbf{b} = \begin{bmatrix} -13 \\ 9 \end{bmatrix}$$

0.5.1 Notation

We use the following notation:

- By $\mathbf{A} \in \mathbb{R}^{m \times n}$, we denote a **matrix** with m rows and n columns, where the entries of A are real numbers.
- By $\mathbf{x} \in \mathbb{R}^n$, we denote a **vector** with n entries. A vector can also be thought of as a matrix with n rows and 1 column, known as a **column vector**. A **row vector** — a matrix with 1 row and n columns is denoted as $\mathbf{x}^T$, the transpose of $\mathbf{x}$.
- The ith element of a vector $\mathbf{x}$ is denoted x_i:

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}.$$

Exercise 0.8 *In the rest of the school we will represent both matrices and vectors as numpy arrays. You can create arrays in different ways, one possible way is to create an array of zeros.*

```
import numpy as np
m = 3
n = 2
a = np.zeros([m,n])
print(a)
[[ 0.   0.]
 [ 0.   0.]
 [ 0.   0.]]
```

You can check the shape and the data type of your array using the following commands:

```
print(a.shape)
(3, 2)
print(a.dtype.name)
float64
```

*This shows you that "a" is an 3*2 array of type float64. By default, arrays contain 64 bit*[6] *floating point numbers. You can specify the particular array type by using the keyword dtype.*

```
a = np.zeros([m,n],dtype=int)
print(a.dtype)
int64
```

You can also create arrays from lists of numbers:

```
a = np.array([[2,3],[3,4]])
print(a)
[[2 3]
 [3 4]]
```

There are many more ways to create arrays in numpy and we will get to see them as we progress in the classes.

0.5.2 Some Matrix Operations and Properties

- Product of two matrices $\mathbf{A} \in \mathbb{R}^{m \times n}$ and $\mathbf{B} \in \mathbb{R}^{n \times p}$ is the matrix $\mathbf{C} = \mathbf{AB} \in \mathbb{R}^{m \times p}$, where

$$C_{ij} = \sum_{k=1}^{n} A_{ik} B_{kj}.$$

Exercise 0.9 *You can multiply two matrices by looping over both indexes and multiplying the individual entries.*

```
a = np.array([[2,3],[3,4]])
b = np.array([[1,1],[1,1]])
a_dim1, a_dim2 = a.shape
b_dim1, b_dim2 = b.shape
c = np.zeros([a_dim1,b_dim2])
for i in range(a_dim1):
   for j in range(b_dim2):
       for k in range(a_dim2):
          c[i,j] += a[i,k]*b[k,j]
print(c)
```

This is, however, cumbersome and inefficient. Numpy supports matrix multiplication with the dot function:

```
d = np.dot(a,b)
print(d)
```

Important note: *with numpy, you must use* `dot` *to get matrix multiplication, the expression a * b denotes element-wise multiplication.*

- Matrix multiplication is associative: $(\mathbf{AB})\mathbf{C} = \mathbf{A}(\mathbf{BC})$.
- Matrix multiplication is distributive: $\mathbf{A}(\mathbf{B} + \mathbf{C}) = \mathbf{AB} + \mathbf{AC}$.

[6] On your computer, particularly if you have an older computer, `int` might denote 32 bits integers

- Matrix multiplication is (generally) not commutative : $\mathbf{AB} \neq \mathbf{BA}$.
- Given two vectors $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n$ the product $\mathbf{x}^T\mathbf{y}$, called **inner product** or **dot product**, is given by

$$\mathbf{x}^T\mathbf{y} \in \mathbb{R} = \begin{bmatrix} x_1 & x_2 & \dots & x_n \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} = \sum_{i=1}^{n} x_i y_i.$$

```
a = np.array([1,2])
b = np.array([1,1])
np.dot(a,b)
```

- Given vectors $\mathbf{x} \in \mathbb{R}^m$ and $\mathbf{y} \in \mathbb{R}^n$, the **outer product** $\mathbf{xy}^T \in \mathbb{R}^{m \times n}$ is a matrix whose entries are given by $(xy^T)_{ij} = x_i y_j$,

$$\mathbf{xy}^T \in \mathbb{R}^{m \times n} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{bmatrix} \begin{bmatrix} y_1 & y_2 & \dots & y_n \end{bmatrix} = \begin{bmatrix} x_1 y_1 & x_1 y_2 & \dots & x_1 y_n \\ x_2 y_1 & x_2 y_2 & \dots & x_2 y_n \\ \vdots & \vdots & \ddots & \vdots \\ x_m y_1 & x_m y_2 & \dots & x_m y_n \end{bmatrix}.$$

```
np.outer(a,b)
array([[1, 1],
       [2, 2]])
```

- The **identity matrix**, denoted $\mathbf{I} \in \mathbb{R}^{n \times n}$, is a square matrix with ones on the diagonal and zeros everywhere else. That is,

$$I_{ij} = \begin{cases} 1 & i = j \\ 0 & i \neq j \end{cases}$$

It has the property that for all $\mathbf{A} \in \mathbb{R}^{n \times n}$, $\mathbf{AI} = \mathbf{A} = \mathbf{IA}$.

```
I = np.eye(2)
x = np.array([2.3, 3.4])

print(I)
print(np.dot(I,x))

[[ 1.,  0.],
 [ 0.,  1.]]
[2.3, 3.4]
```

- A **diagonal matrix** is a matrix where all non-diagonal elements are 0.
- The **transpose** of a matrix results from "'flipping" the rows and columns. Given a matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$, the transpose $\mathbf{A}^T \in \mathbb{R}^{n \times m}$ is the $n \times m$ matrix whose entries are given by $(A^T)_{ij} = A_{ji}$.

Also, $(\mathbf{A}^T)^T = \mathbf{A}$; $\quad (\mathbf{AB})^T = \mathbf{B}^T\mathbf{A}^T$; $\quad (\mathbf{A}+\mathbf{B})^T = \mathbf{A}^T + \mathbf{B}^T$

In numpy, you can access the transpose of a matrix as the `T` attribute:

```
A = np.array([ [1, 2], [3, 4] ])
print(A.T)
```

- A square matrix $\mathbf{A} \in \mathbb{R}^{n \times n}$ is **symmetric** if $\mathbf{A} = \mathbf{A}^T$.
- The **trace** of a square matrix $\mathbf{A} \in \mathbb{R}^{n \times n}$ is the sum of the diagonal elements, $\mathrm{tr}(\mathbf{A}) = \sum_{i=1}^{n} A_{ii}$

0.5.3 Norms

The **norm** of a vector is informally the measure of the "length" of the vector. The commonly used Euclidean or ℓ_2 norm is given by

$$\|\mathbf{x}\|_2 = \sqrt{\sum_{i=1}^{n} x_i^2}.$$

- More generally, the ℓ_p norm of a vector $\mathbf{x} \in \mathbb{R}^n$, where $p \geq 1$ is defined as

$$\|\mathbf{x}\|_p = \left(\sum_{i=1}^{n} |x_i|^p \right)^{1/p}.$$

Note: ℓ_1 norm : $\|\mathbf{x}\|_1 = \sum\limits_{i=1}^{n} |x_i|$ $\quad \ell_\infty$ norm : $\|\mathbf{x}\|_\infty = \max\limits_i |x_i|$.

0.5.4 Linear Independence, Rank, and Orthogonal Matrices

A set of vectors $\{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n\} \subset \mathbb{R}^m$ is said to be **(linearly) independent** if no vector can be represented as a linear combination of the remaining vectors. Conversely, if one vector belonging to the set can be represented as a linear combination of the remaining vectors, then the vectors are said to be **linearly dependent**. That is, if

$$\mathbf{x}_j = \sum_{i \neq j} \alpha_i \mathbf{x}_i$$

for some $j \in \{1, \ldots, n\}$ and some scalar values $\alpha_1, \ldots, \alpha_{i-1}, \alpha_{i+1}, \ldots, \alpha_n \in \mathbb{R}$.

- The **rank** of a matrix is the number of linearly independent columns, which is always equal to the number of linearly independent rows.
- For $\mathbf{A} \in R^{m \times n}$, rank$(\mathbf{A}) \leq \min(m, n)$. If rank$(\mathbf{A}) = \min(m, n)$, then $\mathbf{A}$ is said to be **full rank**.
- For $\mathbf{A} \in R^{m \times n}$, rank$(\mathbf{A})$ = rank$(\mathbf{A}^T)$.
- For $\mathbf{A} \in R^{m \times n}$, $\mathbf{B} \in R^{n \times p}$, rank$(\mathbf{AB}) \leq$ min(rank$(\mathbf{A})$,rank$(\mathbf{B})$).
- For $\mathbf{A}, \mathbf{B} \in R^{m \times n}$, rank$(\mathbf{A} + \mathbf{B}) \leq$ rank$(\mathbf{A})$ + rank$(\mathbf{B})$.
- Two vectors $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n$ are **orthogonal** if $\mathbf{x}^T\mathbf{y} = 0$. A square matrix $\mathbf{U} \in \mathbb{R}^{n \times n}$ is orthogonal if all its columns are orthogonal to each other and are normalized ($\|\mathbf{x}\|_2 = 1$), It follows that

$$\mathbf{U}^T\mathbf{U} = \mathbf{I} = \mathbf{U}\mathbf{U}^T.$$

0.6 Probability Theory

Probability is the most used mathematical language for quantifying uncertainty. The **sample space** $\mathcal{X}$ is the set of possible outcomes of an experiment. **Events** are subsets of $\mathcal{X}$.

Example 0.1 (*discrete space*) *Let H denote "heads" and T denote "tails." If we toss a coin twice, then $\mathcal{X} = \{HH, HT, TH, TT\}$. The event that the first toss is heads is $A = \{HH, HT\}$.*

A sample space can also be *continuous* (eg., $\mathcal{X} = \mathbb{R}$). The union of events A and B is defined as $A \bigcup B = \{\omega \in \mathcal{X} \mid \omega \in A \vee \omega \in B\}$. If $A_1, \ldots, A_n$ is a sequence of sets then $\bigcup\limits_{i=1}^{n} A_i = \{\omega \in \mathcal{X} \mid \omega \in A_i \text{ for at least one i}\}$. We say that $A_1, \ldots, A_n$ are **disjoint** or **mutually exclusive** if $A_i \cap A_j = \varnothing$ whenever $i \neq j$.

We want to assign a real number $P(A)$ to every event A, called the **probability** of A. We also call P a **probability distribution** or **probability measure**.

Definition 0.1 *A function P that assigns a real number $P(A)$ to each event A is a* **probability distribution** *or a* **probability measure** *if it satisfies the three following axioms:*

Axiom 1: $P(A) \geq 0$ for every A
Axiom 2: $P(\mathcal{X}) = 1$
Axiom 3: If $A_1, \ldots, A_n$ are disjoint then

$$P\left(\bigcup_{i=1}^{n} A_i\right) = \sum_{i=1}^{n} P(A_i).$$

One can derive many properties of P from these axioms:

$$\begin{array}{rcl} P(\varnothing) & = & 0 \\ A \subseteq B & \Rightarrow & P(A) \leq P(B) \\ 0 \leq & P(A) & \leq 1 \\ P(A') & = & 1 - P(A) \quad \text{(A' is the complement of A)} \\ P(A \cup B) & = & P(A) + P(B) - P(A \cap B) \\ A \cap B = \phi & \Rightarrow & P\left(A \cup B\right) = P(A) + P(B). \end{array}$$

An important case is when events are **independent**, this is also a usual approximation which lends several practical advantages for the computation of the joint probability.

Definition 0.2 *Two events A and B are* **independent** *if*

$$P(AB) = P(A)P(B) \tag{1}$$

often denoted as $A \perp\!\!\!\perp B$. A set of events $\{A_i : i \in I\}$ is independent if

$$P\left(\bigcap_{i \in J} A_i\right) = \prod_{i \in J} P(A_i)$$

for every finite subset J of I.

For events A and B, where $P(B) > 0$, the **conditional probability** of A given that B has occurred is defined as:

$$P(A|B) = \frac{P(AB)}{P(B)}. \tag{2}$$

Events A and B are independent if and only if $P(A|B) = P(A)$. This follows from the definitions of independence and conditional probability.

A preliminary result that forms the basis for the famous Bayes' theorem is the law of total probability which states that if $A_1, \ldots, A_k$ is a partition of $\mathcal{X}$, then for any event B,

$$P(B) = \sum_{i=1}^{k} P(B|A_i)P(A_i). \tag{3}$$

Using Equations 2 and 3, one can derive the Bayes' theorem.

Theorem 0.1 (Bayes' Theorem) *Let $A_1, \ldots A_k$ be a partition of $\mathcal{X}$ such that $P(A_i) > 0$ for each i. If $P(B) > 0$ then, for each $i = 1, \ldots, k$,*

$$P(A_i|B) = \frac{P(B|A_i)P(A_i)}{\sum_j P(B|A_j)P(A_j)}. \tag{4}$$

Remark 0.1 *$P(A_i)$ is called the* **prior probability** *of A_i and $P(A_i|B)$ is the* **posterior probability** *of A_i.*

Remark 0.2 *In Bayesian Statistical Inference, the Bayes' theorem is used to compute the estimates of distribution parameters from data. Here, prior is the initial* belief *about the parameters, likelihood is the distribution function of the* *parameter (usually trained from data) and posterior is the* updated belief *about the parameters.*

0.6.1 Probability distribution functions

A **random variable** is a mapping $X : \mathcal{X} \to \mathbb{R}$ that assigns a real number $X(\omega)$ to each outcome ω. Given a random variable X, an important function called the **cumulative distributive function** (or **distribution function**) is defined as:

Definition 0.3 *The* **cumulative distribution function** *CDF $F_X : \mathbb{R} \to [0,1]$ of a random variable X is defined by $F_X(x) = P(X \leq x)$.*

The CDF is important because it captures the complete information about the random variable. The CDF is right-continuous, non-decreasing and is normalized ($\lim_{x \to -\infty} F(x) = 0$ and $\lim_{x \to \infty} F(x) = 1$).

Example 0.2 (*discrete CDF*) *Flip a fair coin twice and let X be the random variable indicating the number of heads. Then $P(X = 0) = P(X = 2) = 1/4$ and $P(X = 1) = 1/2$. The distribution function is*

$$F_X(x) = \begin{cases} 0 & x < 0 \\ 1/4 & 0 \leq x < 1 \\ 3/4 & 1 \leq x < 2 \\ 1 & x \geq 2. \end{cases}$$

Definition 0.4 *X is discrete if it takes countable many values $\{x_1, x_2, \ldots\}$. We define the* **probability function** *or* **probability mass function** *for X by*

$$f_X(x) = P(X = x).$$

Definition 0.5 *A random variable X is* **continuous** *if there exists a function f_X such that $f_X \geq 0$ for all x, $\int\limits_{-\infty}^{\infty} f_X(x)dx = 1$ and for every $a \leq b$*

$$P(a < X < b) = \int\limits_a^b f_X(x)dx. \tag{5}$$

The function f_X is called the **probability density function** *(PDF). We have that*

$$F_X(x) = \int\limits_{-\infty}^{x} f_X(t)dt$$

and $f_X(x) = F_X^{'}(x)$ at all points x at which F_X is differentiable.

A discussion of a few important distributions and related properties:

0.6.2 Bernoulli

The **Bernoulli distribution** is a discrete probability distribution that takes value 1 with the success probability p and 0 with the failure probability $q = 1 - p$. A single Bernoulli trial is parametrized with the success probability p, and the input $k \in \{0, 1\}$ (1=success, 0=failure), and can be expressed as

$$f(k; p) = p^k q^{1-k} = p^k (1-p)^{1-k}$$

0.6.3 Binomial

The probability distribution for the number of successes in n Bernoulli trials is called a **Binomial distribution**, which is also a discrete distribution. The Binomial distribution can be expressed as exactly j successes is

$$f(j, n; p) = \left(\begin{array}{c} n \\ j \end{array} \right) p^j q^{n-j} = \left(\begin{array}{c} n \\ j \end{array} \right) p^j (1-p)^{n-j}$$

where n is the number of Bernoulli trials with probability p of success on each trial.

0.6.4 Categorical

The **Categorical distribution** (often conflated with the Multinomial distribution, in fields like Natural Language Processing) is another generalization of the Bernoulli distribution, allowing the definition of a set of possible outcomes, rather than simply the events "success" and "failure" defined in the Bernoulli distribution. Considering a set of outcomes indexed from 1 to n, the distribution takes the form of

$$f(x_i; p_1, ..., p_n) = p_i.$$

where parameters $p_1, ..., p_n$ is the set with the occurrence probability of each outcome. Note that we must ensure that $\sum_{i=1}^{n} p_i = 1$, so we can set $p_n = 1 - \sum_{i=1}^{n-1} p_i$.

0.6.5 Multinomial

The **Multinomial distribution** is a generalization of the Binomial distribution and the Categorical distribution, since it considers multiple outcomes, as the Categorial distribution, and multiple trials, as in the Binomial distribution. Considering a set of outcomes indexed from 1 to n, the vector $[x_1, ..., x_n]$, where x_i indicates the number of times the event with index i occurs, follows the Multinomial distribution

$$f(x_1, ..., x_n; p_1, ..., p_n) = \frac{n!}{x_1!...x_n!} p_1^{x_1}...p_n^{x_n}.$$

Where parameters $p_1, ..., p_n$ represent the occurrence probability of the respective outcome.

0.6.6 Gaussian Distribution

A very important theorem in probability theory is the **Central Limit Theorem**. The Central Limit Theorem states that, under very general conditions, if we sum a very large number of mutually independent random variables, then the distribution of the sum can be closely approximated by a certain specific continuous density called the normal (or Gaussian) density. The normal density function with parameters μ and σ is defined as follows:

$$f_X(x) = \frac{1}{\sqrt{2\pi}\sigma} e^{-(x-\mu)^2/2\sigma^2}, \ -\infty < x < \infty.$$

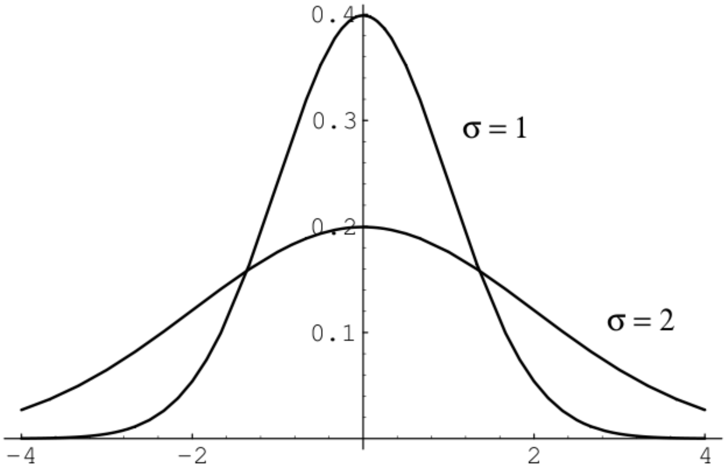


Figure 1: Normal density for two sets of parameter values.

Figure 1 compares a plot of normal density for the cases $\mu = 0$ and $\sigma = 1$, and $\mu = 0$ and $\sigma = 2$.

0.6.7 Maximum Likelihood Estimation

Until now we assumed that, for every distribution, the parameters θ are known and are used when we calculate $p(x|\theta)$. There are some cases where the values of the parameters are easy to infer, such as the probability p of getting a head using a fair coin, used on a Bernoulli or Binomial distribution. However, in many problems, these values are complex to define and it is more viable to estimate the parameters using the data x. For instance, in the example above with the coin toss, if the coin is somehow tampered to have a biased behavior, rather than examining the dynamics or the structure of the coin to infer a parameter for p, a person could simply throw the coin n times, count the number of heads h and set $p = \frac{h}{n}$. By doing so, the person is using the data x to estimate θ.

With this in mind, we will now generalize this process by defining the probability $p(\theta|x)$ as the probability of the parameter θ, given the data x. This probability is called **likelihood** $\mathcal{L}(\theta|x)$ and measures how well the parameter θ models the data x. The likelihood can be defined in terms of the distribution f as

$$\mathcal{L}(\theta|x_1, ..., x_n) = \prod_{i=1}^{n} f(x_i|\theta)$$

where $x_1, ..., x_n$ are independently and identically distributed (i.i.d.) samples.

To understand this concept better, we go back to the tampered coin example again. Suppose that we throw the coin 5 times and get the sequence [1,1,1,1,1] (1=head, 0=tail). Using the Bernoulli distribution (see Section 0.6.2) f to model this problem, we get the following likelihood values:

- $\mathcal{L}(0, x) = f(1, 0)^5 = 0^5 = 0$
- $\mathcal{L}(0.2, x) = f(1, 0.2)^5 = 0.2^5 = 0.00032$
- $\mathcal{L}(0.4, x) = f(1, 0.4)^5 = 0.4^5 = 0.01024$
- $\mathcal{L}(0.6, x) = f(1, 0.6)^5 = 0.6^5 = 0.07776$
- $\mathcal{L}(0.8, x) = f(1, 0.8)^5 = 0.8^5 = 0.32768$
- $\mathcal{L}(1, x) = f(1, 1)^5 = 1^5 = 1$

If we get the sequence [1,0,1,1,0] instead, the likelihood values would be:

- $\mathcal{L}(0, x) = f(1, 0)^3 f(0, 0)^2 = 0^3 \times 1^2 = 0$
- $\mathcal{L}(0.2, x) = f(1, 0.2)^3 f(0, 0.2)^2 = 0.2^3 \times 0.8^2 = 0.00512$
- $\mathcal{L}(0.4, x) = f(1, 0.4)^3 f(0, 0.4)^2 = 0.4^3 \times 0.6^2 = 0.02304$
- $\mathcal{L}(0.6, x) = f(1, 0.6)^3 f(0, 0.6)^2 = 0.6^3 \times 0.4^2 = 0.03456$
- $\mathcal{L}(0.8, x) = f(1, 0.8)^3 f(0, 0.8)^2 = 0.8^3 \times 0.2^2 = 0.02048$
- $\mathcal{L}(1, x) = f(1, 1)^5 = 1^3 \times 0^2 = 0$

We can see that the likelihood is the highest when the distribution f with parameter p is the best fit for the observed samples. Thus, the best estimate for p according to x would be the value for which $\mathcal{L}(p|x)$ is the highest.

The value of the parameter θ with the highest likelihood is called **maximum likelihood estimate (MLE)** and is defined as

$$\hat{\theta}_{mle} = argmax_{\theta} \mathcal{L}(\theta|x)$$

Finding this for our example is relatively easy, since we can simply derivate the likelihood function to find the absolute maximum. For the sequence [1,0,1,1,0], the likelihood would be given as

$$\mathcal{L}(p|x) = f(1, p)^3 f(0, p)^2 = p^3(1 - p)^2$$

And the MLE estimate would be given by:

$$\frac{\delta \mathcal{L}(p|x)}{\delta p} = 0,$$

which resolves to

$$p_{mle} = 0.6$$

Exercise 0.10 *Over the next couple of exercises we will make use of the Galton dataset, a dataset of heights of fathers and sons from the 1877 paper that first discussed the "regression to the mean" phenomenon. This dataset has 928 pairs of numbers.*

- *Use the* `load()` *function in the* `galton.py` *file to load the dataset. The file is located under the* `lxmls/readers` *folder*[7]*. Type the following in your Python interpreter:*

  ```
  import galton as galton
  galton_data = galton.load()
  ```

- *What are the mean height and standard deviation of all the people in the sample? What is the mean height of the fathers and of the sons?*
- *Plot a histogram of all the heights (you might want to use the* `plt.hist` *function and the* `ravel` *method on arrays).*
- *Plot the height of the father versus the height of the son.*
- *You should notice that there are several points that are exactly the same (e.g., there are 21 pairs with the values 68.5 and 70.2). Use the* `?` *command in ipython to read the documentation for the* `numpy.random.randn` *function and add random jitter (i.e., move the point a little bit) to the points before displaying them. Does your impression of the data change?*

0.6.8 Conjugate Priors

Definition 0.6 *let $\mathcal{F} = \{f_X(x|s), s \in \mathcal{X}\}$ be a class of likelihood functions; let $\mathcal{P}$ be a class of probability (density or mass) functions; if, for any x, any $p_S(s) \in \mathcal{P}$, and any $f_X(x|s) \in \mathcal{F}$, the resulting a posteriori probability function $p_S(s|x) = f_X(x|s)p_S(s)$ is still in $\mathcal{P}$, then $\mathcal{P}$ is called a conjugate family, or a family of* **conjugate priors***, for $\mathcal{F}$.*

0.7 Numerical optimization

Most problems in machine learning require minimization/maximization of functions (likelihoods, risk, energy, entropy, etc.,). Let x^* be the value of x which minimizes the value of some function $f(x)$. Mathematically, this is written as

$$x^* = \arg\min_x f(x)$$

In a few special cases, we can solve this minimization problem analytically in closed form (solving for optimal x^* in $\nabla_x f(x^*) = 0$), but in most cases it is too cumbersome (or impossible) to solve these equations analytically, and they must be tackled numerically. In this section we will cover some basic notions of numerical optimization. The goal is to provide the intuitions behind the methods that will be used in the rest of the school. There are plenty of good textbooks in the subject that you can consult for more information (Nocedal and Wright, 1999; Bertsekas et al., 1995; Boyd and Vandenberghe, 2004).

The most common way to solve the problems when no closed form solution is available is to resort to an iterative algorithm. In this Section, we will see some of these iterative optimization techniques. These iterative algorithms construct a sequence of points $x^{(0)}, x^{(1)}, \ldots \in \text{domain}(f)$ such that hopefully $x^t = x^*$ after a number of iterations. Such a sequence is called the **minimizing sequence** for the problem.

0.7.1 Convex Functions

One important property of a function $f(x)$ is whether it is a **convex function** (in the shape of a bowl) or a **non-convex function**. Figures 2 and 3 show an example of a convex and a non-convex function. Convex functions are particularly useful since you can guarantee that the minimizing sequence converges to the true global minimum of the function, while in non-convex functions you can only guarantee that it will reach a local minimum.

Intuitively, imagine dropping a ball on either side of Figure 2, the ball will roll to the bottom of the bowl independently from where it is dropped. This is the main benefit of a convex function. On the other hand, if

[7] You might need to inform python about the location of the lxmls labs toolkit. To do so you need to `import sys` and use the `sys.path.append` function to add the path to the toolkit readers.

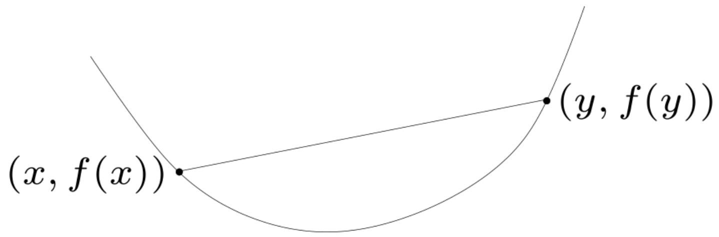


Figure 2: Illustration of a convex function. The line segment between any two points on the graph lies entirely above the curve.

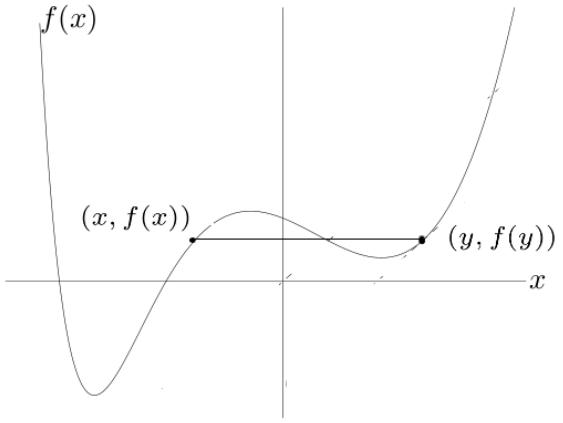


Figure 3: Illustration of a non-convex function. Note the line segment intersecting the curve.

you drop a ball from the left side of Figure 3 it will reach a different position than if you drop a ball from its right side. Moreover, dropping it from the left side will lead you to a much better (*i.e.*, lower) place than if you drop the ball from the right side. This is the main problem with non-convex functions: there are no guarantees about the quality of the local minimum you find.

More formally, some concepts to understand about convex functions are:
A **line segment** between points x_1 and x_2: contains all points such that

$$x = \theta x_1 + (1 - \theta)x_2$$

where $0 \leq \theta \leq 1$.

A **convex set** contains the line segment between any two points in the set

$$x_1, x_2 \in C, \quad 0 \leq \theta \leq 1 \quad \Rightarrow \quad \theta x_1 + (1 - \theta)x_2 \in C.$$

A function $f : \mathbb{R}^n \rightarrow R$ is a **convex function** if the domain of f is a convex set and

$$f(\theta \mathbf{x} + (1 - \theta)\mathbf{y}) \leq \theta f(\mathbf{x}) + (1 - \theta)f(\mathbf{y})$$

for all $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n$, $0 \leq \theta \leq 1$

0.7.2 Derivative and Gradient

The **derivative** of a function is a measure of how the function varies with its input variables. Given an interval $[a, b]$ one can compute how the function varies within that interval by calculating the average slope of the function in that interval:

$$\frac{f(b) - f(a)}{b - a}. \tag{6}$$

The derivative can be seen as the limit as the interval goes to zero, and it gives us the slope of the function at that point.

$$\frac{\partial f}{\partial x} = \lim_{h=0} \frac{f(x+h) - f(x)}{h} \tag{7}$$

Table 3 shows derivatives of some functions that we will be using during the school.

Function $f(x)$	Derivative $\frac{\partial f}{\partial x}$
x^2	$2x$
x^n	nx^{n-1}
$\log(x)$	$\frac{1}{x}$
$\exp(x)$	$\exp(x)$
$\frac{1}{x}$	$-\frac{1}{x^2}$

Table 3: Some derivative examples

An important rule of derivation is the chain rule. Consider $h = f \circ g$, and $u = g(x)$, then:

$$\frac{\partial h}{\partial x} = \frac{\partial f}{\partial u} \cdot \frac{\partial g}{\partial x} \tag{8}$$

Example 0.3 *Consider the function* $h(x) = \exp(x^2)$*, this can be decomposed as* $h(x) = f(g(x)) = f(u) = \exp(u)$*, where* $u = g(x) = x^2$ *and has derivative* $\frac{\partial h}{\partial x} = \frac{\partial f}{\partial u} \cdot \frac{\partial u}{\partial x} = \exp(u) \cdot 2x = \exp(x^2) \cdot 2x$

Exercise 0.11 *Consider the function* $f(x) = x^2$ *and its derivative* $\frac{\partial f}{\partial x}$*. Look at the derivative of that function at points [-2,0,2], draw the tangent to the graph in that point* $\frac{\partial f}{\partial x}(-2) = -4$*,* $\frac{\partial f}{\partial x}(0) = 0$*, and* $\frac{\partial f}{\partial x}(2) = 4$*. For example, the tangent equation for* $x = -2$ *is* $y = -4x - b$*, where* $b = f(-2)$*. The following code plots the function and the derivatives on those points using matplotlib (See Figure 4).*

```
a = np.arange(-5,5,0.01)
f_x = np.power(a,2)
plt.plot(a,f_x)

plt.xlim(-5,5)
plt.ylim(-5,15)

k= np.array([-2,0,2])
plt.plot(k,k**2,"bo")
for i in k:
    plt.plot(a, (2*i)*a - (i**2))
```

The **gradient** of a function is a generalization of the derivative concept we just saw before for several dimensions. Let us assume we have a function $f(\mathbf{x})$ where $\mathbf{x} \in \mathbb{R}^2$, so $\mathbf{x}$ can be seen as a pair $\mathbf{x} = [x_1, x_2]$. Then, the gradient measures the slope of the function in both directions: $\nabla_{\mathbf{x}} f(\mathbf{x}) = [\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}]$.

0.7.3 Gradient Based Methods

Gradient based methods are probably the most common methods used for finding the minimizing sequence for a given function. The methods used in this class will make use of the function value $f(\mathbf{x}) : \mathbb{R}^n \rightarrow R$ as well as the gradient of the function $\nabla_{\mathbf{x}} f(\mathbf{x})$. The simplest method is the **Gradient descent** method, an unconstrained first-order optimization algorithm.

The intuition of this method is as follows: You start at a given point $\mathbf{x}_0$ and compute the gradient at that point $\nabla_{\mathbf{x}_0} f(\mathbf{x})$. You then take a step of length η on the direction of the negative gradient to find a new point: $\mathbf{x}_1 = \mathbf{x}_0 - \eta \nabla_{\mathbf{x}_0} f(\mathbf{x})$. Then, you compute the gradient at this new point, $\nabla_{\mathbf{x}_1} f(\mathbf{x})$, and take a step of length η on the direction of the negative gradient to find a new point: $\mathbf{x}_2 = \mathbf{x}_1 - \eta \nabla_{\mathbf{x}_1} f(\mathbf{x})$. You proceed until you have reached a minimum (local or global). Recall from the previous subsection that you can identify the minimum by testing if the norm of the gradient is zero: $||\nabla f(\mathbf{x})|| = 0$.

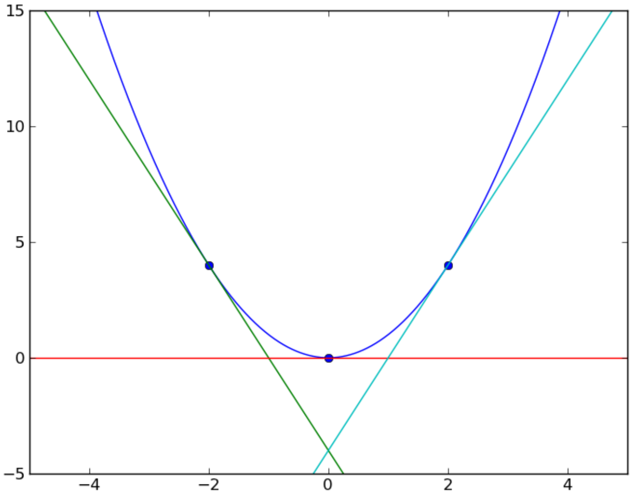


Figure 4: Illustration of the gradient of the function $f(x^2)$ at three different points $x = [-2, 0, 2]$. Note that at point $x = 0$ the gradient is zero which corresponds to the minimum of the function.

There are several practical concerns even with this basic algorithm to ensure both that the algorithm converges (reaches the minimum) and that it does so in a fast way (by fast we mean the number of function and gradient evaluations).

- **Step Size** η A first question is how to find the step length η. One condition is that *eta* should guarantee sufficient decrease in the function value. We will not cover these methods here but the most common ones are **Backtracking line search** or the **Wolf Line Search** (Nocedal and Wright, 1999).

- **Descent Direction** A second problem is that using the negative gradient as direction can lead to a very slow convergence. Different methods that change the descent direction by multiplying the gradient by a matrix **B** have been proposed that guarantee a faster convergence. Two notable methods are the Conjugate Gradient (CG) and the Limited Memory Quasi Newton methods (LBFGS) (Nocedal and Wright, 1999).

- **Stopping Criteria** Finally, it will normally not be possible to reach full convergence either because it will be too slow, or because of numerical issues (computers cannot perform exact arithmetic). So normally we need to define a stopping criteria for the algorithm. Three common criteria (that are normally used together) are: a maximum number of iterations; the gradient norm be smaller than a given threshold $||\nabla f(\mathbf{x})|| \leq \eta_1$, or the normalized difference in the function value be smaller than a given threshold $\frac{|f(\mathbf{x}_t)-f(\mathbf{x}_{t-1})|}{\max(|f(\mathbf{x}_t)|,|f(\mathbf{x}_{t-1})|)} \leq \eta_2$

Algorithm 1 shows the general gradient based algorithm. Note that for the simple gradient descent algorithm **B** is the identity matrix and the descent direction is just the negative gradient of the function.

Algorithm 1 Gradient Descent

```
given a starting point $\mathbf{x}_0, i = 0$
repeat
    Compute step size $\eta$
    Compute descent direction $-\mathbf{B}\nabla f(x_i)$.
    $\mathbf{x}_{i+1} \leftarrow \mathbf{x}_i - \eta \mathbf{B} \nabla f(\mathbf{x}_i)$
    $i \leftarrow i + 1$
until stopping criterion is satisfied.
```

Exercise 0.12 *Consider the function* $f(x) = (x+2)^2 - 16\exp\left(-(x-2)^2\right)$. *Make a function that computes the function value given* x.

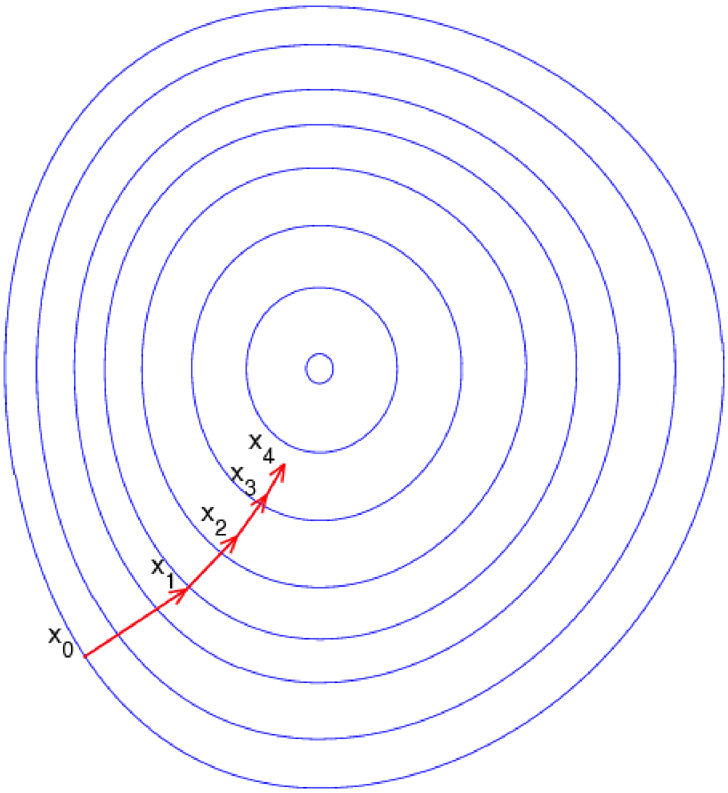


Figure 5: Illustration of gradient descent. The blue circles correspond to contours of the function (each blue circle is a set of points which have the same function value), while the red lines correspond to steps taken in the negative gradient direction.

```
def get_y(x):
    return (x+2)**2 - 16*np.exp(-((x-2)**2))
```

Draw a plot around $x \in [-8, 8]$.

```
x = np.arange(-8, 8, 0.001)
y = get_y(x)
plt.plot(x, y)
plt.show()
```

Calculate the derivative of the function $f(x)$, *implement the function* get_grad(x).

```
def get_grad(x):
    return (2*x+4)-16*(-2*x + 4)*np.exp(-((x-2)**2))
```

Use the method gradient_descent *to find the minimum of this function. Convince yourself that the code is doing the proper thing. Look at the constants we defined. Note, that we are using a simple approach to pick the step size (always have the value step_size) which is not necessarily correct.*

```
def gradient_descent_scalar(start_x, func, grad, step_size=0.1, prec=0.0001):
    max_iter=100
    x_new = start_x
    res = []
    for i in range(max_iter):
        x_old = x_new
        # Use negative step size for gradient descent
        x_new = x_old - step_size * grad(x_new)
        f_x_new = func(x_new)
        f_x_old = func(x_old)
        res.append([x_new,f_x_new])

        if(abs(f_x_new - f_x_old) < prec):
            print("change in function values too small, leaving")
            return np.array(res)
    print("exceeded maximum number of iterations, leaving")
    return np.array(res)
```

Run the gradient descent algorithm starting from $x_0 = -8$ and plot the minimizing sequence.

```
x = np.arange(-8, 8, 0.001)
y = get_y(x)
plt.plot(x, y)

x_0 = -8
res = gradient_descent_scalar(x_0, get_y, get_grad)
plt.plot(res[:,0], res[:,1], 'r+')
plt.show()
```

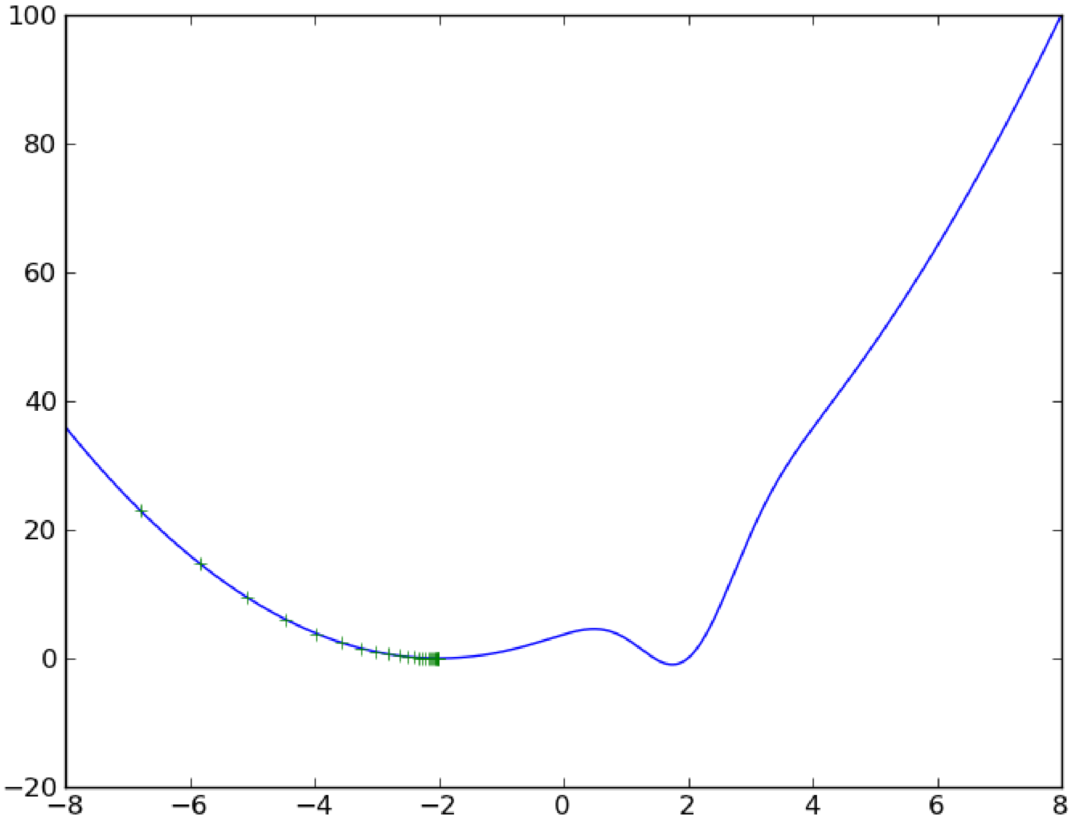


Figure 6: Example of running gradient descent starting on point $x_0 = -8$ for function $f(x) = (x+2)^2 - 16\exp\left(-(x-2)^2\right)$. The function is represented in blue, while the points of the minimizing sequence are displayed as green plus signs.

Figure 6 shows the resulting minimizing sequence. Note that the algorithm converged to a minimum, but since the function is not convex it converged only to a local minimum.

Now try the same exercise starting from the initial point $x_0 = 8$.

```
x = np.arange(-8, 8, 0.001)
y = get_y(x)
plt.plot(x, y)

x_0 = 8
res = gradient_descent_scalar(x_0, get_y, get_grad)
plt.plot(res[:,0], res[:,1], 'r+')
plt.show()
```

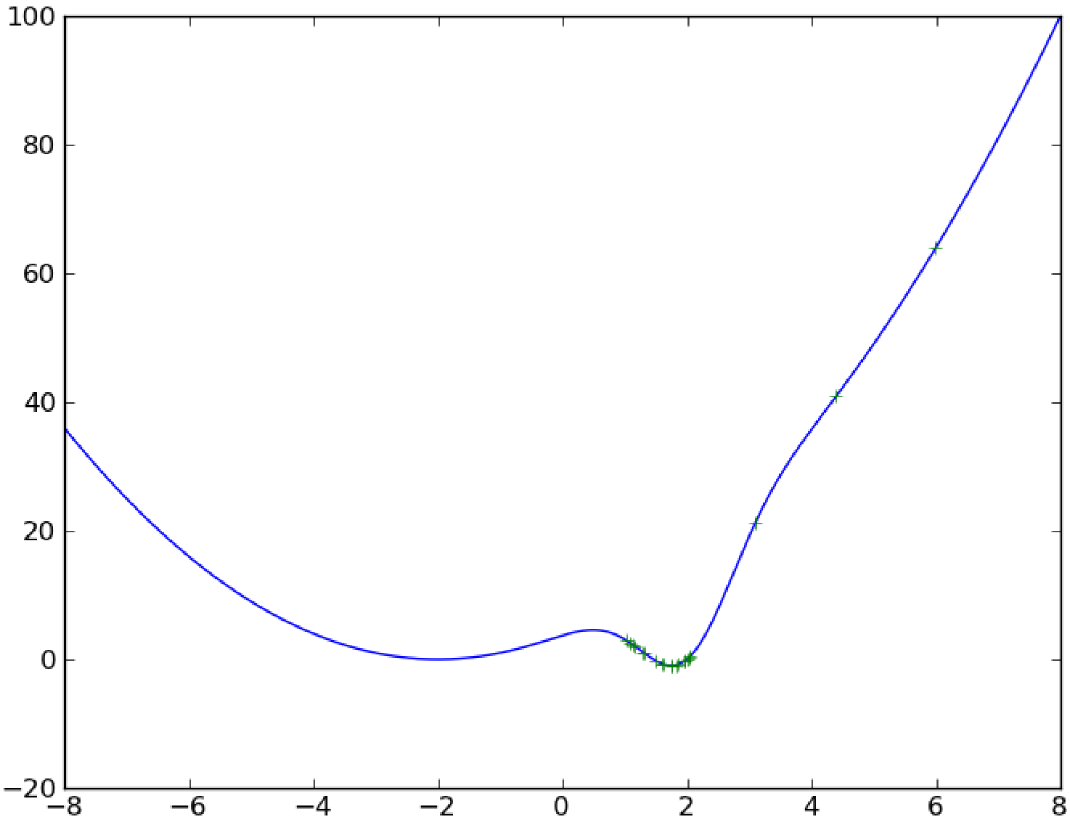


Figure 7: Example of running gradient descent starting on point $x_0 = 8$ for function $f(x) = (x+2)^2 - 16\exp\left(-(x-2)^2\right)$. The function is represented in blue, while the points of the minimizing sequence are displayed as green plus signs.

Figure 7 shows the resulting minimizing sequence. Note that now the algorithm converged to the global minimum. However, note that to get to the global minimum the sequence of points jumped from one side of the minimum to the other. This is a consequence of using a wrong step size (in this case too large). Repeat the previous exercise changing both the values of the step-size and the precision. What do you observe?

During this school we will rely on the numerical optimization methods provided by Scipy (scientific computing library in python), which are very efficient implementations.

0.8 Python Exercises

0.8.1 Numpy and Matplotlib

Exercise 0.13 *Consider the linear regression problem (ordinary least squares) on the Galton dataset, with a single response variable*

$$y = x^T w + \varepsilon$$

.

The linear regression problem *is, given a set* $\{y^{(i)}\}_i$ *of samples of y and the corresponding* $\boldsymbol{x}^{(i)}$ *vectors, estimate* $\boldsymbol{w}$ *to minimise the sum of the ε variables. Traditionally this is solved analytically to obtain a closed form solution (although this is* ***not the way in which it should be computed*** *in this exercise, linear algebra packages have an optimised solver, with numpy, use* `numpy.linalg.lstsq`*).*

Alternatively, we can define the error function for each possible $\boldsymbol{w}$*:*

$$e(\boldsymbol{w}) = \sum_i \left({\boldsymbol{x}^{(i)}}^T \boldsymbol{w} - y^{(i)} \right)^2 .$$

1. *Derive the gradient of the error* $\frac{\partial e}{\partial w_j}$.
2. *Implement a solver based on this for two dimensional problems (i.e.,* $\boldsymbol{w} \in R^2$*).*
3. *Use this method on the Galton dataset from the previous exercise to estimate the relationship between father and son's height. Try two formulas*
$$s = f w_1 + \varepsilon, \tag{9}$$
where s is the son's height, and f is the father heights; and
$$s = f w_1 + 1 w_0 + \varepsilon \tag{10}$$
where the input variable is now two dimensional: $(f, 1)$*. This allows the intercept to be non-zero.*
4. *Plot the regression line you obtain with the points from the previous exercise.*
5. *Use the* `np.linalg.lstsq`*function and compare to your solution.*

Please refer to the notebook for solutions.

0.8.2 Debugging

Exercise 0.14 *Use the debugger to debug the* `buggy.py` *script which attempts to repeatedly perform the following computation:*

1. *Start* $x_0 = 0$
2. *Iterate*
   (a) $x'_{t+1} = x_t + r$, *where r is a random variable.*
   (b) *if* $x'_{t+1} >= 1.$, *then stop.*
   (c) *if* $x'_{t+1} <= 0.$, *then* $x_{t+1} = 0$
   (d) *else* $x_{t+1} = x'_{t+1}$.
3. *Return the number of iterations.*

Having repeated this computation a number of times, the programme prints the average. Unfortunately, the program has a few bugs, which you need to fix.

Day 1

Linear Classifiers

This day will serve as an introduction to machine learning. We will recall some fundamental concepts about decision theory and classification, present some widely used models and algorithms and try to provide the main motivation behind them. There are several textbooks that provide a thorough description of some of the concepts introduced here: for example, Mitchell (1997), Duda et al. (2001), Schölkopf and Smola (2002), Joachims (2002), Bishop (2006), Manning et al. (2008), to name just a few. The concepts that we introduce in this chapter will be revisited in later chapters and expanded to account for non-linear models and structured inputs and outputs. For now, we will concern ourselves only with multi-class classification (with just a few classes) and linear classifiers.

Today's assignment

The assignment of today's class is to implement a classifier called Naïve Bayes, and use it to perform sentiment analysis on a corpus of book reviews from Amazon.

1.1 Notation

In what follows, we denote by $\mathcal{X}$ our *input set* (also called *observation set*) and by $\mathcal{Y}$ our *output set*. We will make no assumptions about the set $\mathcal{X}$, which can be continuous or discrete. In this lecture, we consider *classification* problems, where $\mathcal{Y} = \{c_1, \ldots, c_K\}$ is a finite set, consisting of K *classes* (also called *labels*). For example, $\mathcal{X}$ can be a set of documents in natural language, and $\mathcal{Y}$ a set of topics, the goal being to assign a topic to each document.

We use upper-case letters for denoting random variables, and lower-case letters for value assignments to those variables: for example,

- X is a random variable taking values on $\mathcal{X}$,
- Y is a random variable taking values on $\mathcal{Y}$,
- $x \in \mathcal{X}$ and $y \in \mathcal{Y}$ are particular values for X and Y.

We consider *events* such as $X = x$, $Y = y$, etc.

For simplicity reasons, throughout this lecture we will use modified notation and let $P(y)$ denote the *probability* associated with the event $Y = y$ (instead of $P_Y(Y = y)$). Also, *joint* and *conditional* probabilities are denoted as $P(x, y) \triangleq P_{X,Y}(X = x \wedge Y = y)$ and $P(x|y) \triangleq P_{X|Y}(X = x \mid Y = y)$, respectively. From the laws of probabilities:

$$P(x, y) = P(y|x)P(x) = P(x|y)P(y), \tag{1.1}$$

for all $x \in \mathcal{X}$ and $y \in \mathcal{Y}$.

Quantities that are predicted or estimated from the data will be appended a hat-symbol: for example, estimations of the probabilities above are denoted as $\hat{P}(y)$, $\hat{P}(x, y)$ and $\hat{P}(y|x)$; and a prediction of an output will be denoted $\hat{y}$.

We assume that a *training dataset* $\mathcal{D}$ is provided which consists of M input-output pairs (called *examples* or *instances*):

$$\mathcal{D} = \{(x^1, y^1), \ldots, (x^M, y^M)\} \subseteq \mathcal{X} \times \mathcal{Y}. \tag{1.2}$$

The **goal of (supervised) machine learning** is to use the training dataset $\mathcal{D}$ to learn a function h (called a *classifier*) that maps from $\mathcal{X}$ to $\mathcal{Y}$: this way, given a new instance $x \in \mathcal{X}$ (test example), the machine makes a prediction $\hat{y}$ by evaluating h on x, i.e., $\hat{y} = h(x)$.

1.2 Generative Classifiers: Naïve Bayes

If we knew the *true* distribution $P(X, Y)$, the best possible classifier (called Bayes optimal) would be one which predicts according to

$$\begin{aligned}
\hat{y} &= \arg\max_{y \in \mathcal{Y}} P(y|x) \\
&= \arg\max_{y \in \mathcal{Y}} \frac{P(x,y)}{P(x)} \\
&=^{\dagger} \arg\max_{y \in \mathcal{Y}} P(x,y) \\
&= \arg\max_{y \in \mathcal{Y}} P(y)P(x|y), \qquad (1.3)
\end{aligned}$$

where in † we used the fact that $P(x)$ is constant with respect to y.

Generative classifiers try to estimate the probability distributions $P(Y)$ and $P(X|Y)$, which are respectively called the *class prior* and the *class conditionals*. They assume that the data are generated according to the following generative story (independently for each $m = 1, \ldots, M$):

1. A class $y_m \sim P(Y)$ is drawn from the class prior distribution;
2. An input $x_m \sim P(X|Y = y_m)$ is drawn from the corresponding class conditional.

Figure 1.1 shows an example of the Bayes optimal decision boundary for a toy example with $K = 2$ classes, $M = 100$ points, class priors $P(y_1) = P(y_2) = 0.5$, and class conditionals $P(x|y_i)$ given by 2-D Gaussian distributions with the same variance but different means.

1.2.1 Training and Inference

Training a generative model amounts to *estimating* the probabilities $P(Y)$ and $P(X|Y)$ using the dataset $\mathcal{D}$, yielding estimates $\hat{P}(y)$ and $\hat{P}(x|y)$. This estimation is usually called *training* or *learning*.

After we are done training, we are given a new input $x \in \mathcal{X}$, and we want to make a prediction according to

$$\hat{y} = \arg\max_{y \in \mathcal{Y}} \hat{P}(y)\hat{P}(x|y), \qquad (1.4)$$

using the estimated probabilities. This is usually called *inference* or *decoding*.

We are left with two important problems:

1. How should the distributions $\hat{P}(Y)$ and $\hat{P}(X|Y)$ be "defined"? (i.e., what kind of independence assumptions should they state, or how should they factor?)
2. How should parameters be estimated from the training data $\mathcal{D}$?

The first problem strongly depends on the application at hand. Quite often, there is a natural decomposition of the input variable X into J components,

$$X = (X_1, \ldots, X_J). \qquad (1.5)$$

The naïve Bayes method makes the following assumption: $X_1, \ldots, X_J$ *are conditionally independent given the class*. Mathematically, this means that

$$P(X|Y) = \prod_{j=1}^{J} P(X_j|Y). \qquad (1.6)$$

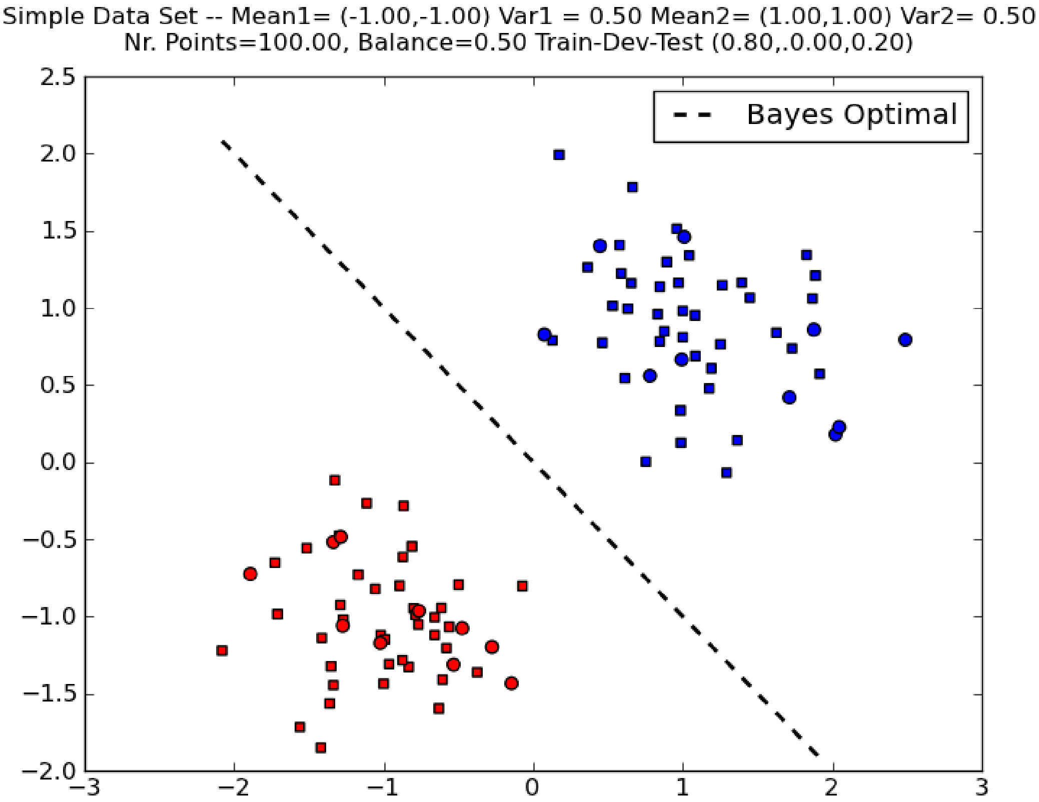


Figure 1.1: Example of a dataset together with the corresponding Bayes optimal decision boundary. The input set consists in points in the real plane, $\mathcal{X} = \mathcal{R}$, and the output set consists of two classes (Red and Blue). Training points are represented as squares, while test points are represented as circles.

Note that this independence assumption greatly reduces the number of parameters to be estimated (degrees of freedom) from $O(\exp(J))$ to $O(J)$, hence estimation of $\hat{P}(X|Y)$ becomes much simpler, as we shall see. It also makes the overall computation much more efficient for large J and it decreases the risk of overfitting the data. On the other hand, if the assumption is over-simplistic it may increase the risk of under-fitting.

For the second problem, one of the simplest ways to solve it is using *maximum likelihood estimation*, which aims to maximize the probability of the training sample, assuming that each point was generated independently. This probability (call it $P(\mathcal{D})$) factorizes as

$$\begin{aligned} P(\mathcal{D}) &= \prod_{m=1}^{M} P(x^m, y^m) \\ &= \prod_{m=1}^{M} P(y^m) \prod_{j=1}^{J} P(x_j^m | y^m). \end{aligned} \tag{1.7}$$

1.2.2 Example: Multinomial Naïve Bayes for Document Classification

We now consider a more realistic scenario where the naïve Bayes classifier may be applied. Suppose that the task is *document classification*: $\mathcal{X}$ is the set of all possible documents, and $\mathcal{Y} = \{y_1, \ldots, y_K\}$ is a set of classes for those documents. Let $\mathcal{V} = \{w_1, \ldots, w_J\}$ be the vocabulary, i.e., the set of words that occur in some document.

A very popular document representation is through a "bag-of-words": each document is seen as a collection of words along with their frequencies; word ordering is ignored. We are going to see that this is equivalent to a naïve Bayes assumption with the *multinomial model*. We associate to each class a multinomial distribution, which ignores word ordering, but takes into consideration the frequency with which each word appears in a document. For simplicity, we assume that all documents have the same length L.[1] Each document x is as-

[1] We can get rid of this assumption by defining a distribution on the document length. Everything stays the same if that distribution is uniform up to a maximum document length.

sumed to have been generated as follows. First, a class y is generated according to $P(y)$. Then, x is generated by sequentially picking words from $\mathcal{V}$ with replacement. Each word w_j is picked with probability $P(w_j|y)$. For example, the probability of generating a document $x = w_{j_1} \dots w_{j_L}$ (*i.e.*, a sequence of L words $w_{j_1}, \dots, w_{j_L}$) is

$$P(x|y) = \prod_{l=1}^{L} P(w_{j_l}|y) = \prod_{j=1}^{J} P(w_j|y)^{n_j(x)}, \tag{1.8}$$

where $n_j(x)$ is the number of occurrences of word w_j in document x.

Hence, the assumption is that word occurrences (*tokens*) are independent given the class. The parameters that need to be estimated are $\hat{P}(y_1), \dots, \hat{P}(y_K)$, and $\hat{P}(w_j|y_k)$ for $j = 1, \dots, J$ and $k = 1, \dots, K$. Given a training sample $\mathcal{D} = \{(x^1, y^1), \dots, (x^M, y^M)\}$, denote by $\mathcal{I}_k$ the indices of those instances belonging to the kth class. The maximum likelihood estimates of the quantities above are:

$$\hat{P}(y_k) = \frac{|\mathcal{I}_k|}{M}, \qquad \hat{P}(w_j|y_k) = \frac{\sum_{m \in \mathcal{I}_k} n_j(x^m)}{\sum_{i=1}^{J} \sum_{m \in \mathcal{I}_k} n_i(x^m)}. \tag{1.9}$$

In words: the class priors' estimates are their relative frequencies (as before), and the class-conditional word probabilities are the relative frequencies of those words across documents with that class.

1.3 Assignment

With the previous theoretical background, you will be able to solve today's assignment.

Exercise 1.1 *In this exercise we will use the Amazon sentiment analysis data (Blitzer et al., 2007), where the goal is to classify text documents as expressing a* positive *or* negative *sentiment (i.e., a classification problem with two classes). We are going to focus on book reviews. To load the data, type:*

```
import lxmls.readers.sentiment_reader as srs

scr = srs.SentimentCorpus("books")
```

This will load the data in a bag-of-words representation where rare words (occurring less than 5 *times in the training data) are removed.*

1. *Implement the Naïve Bayes algorithm. Open the file* `multinomial_naive_bayes.py`*, which is inside the* `classifiers` *folder. In the* `MultinomialNaiveBayes` *class you will find the* `train` *method. We have already placed some code in that file to help you get started.*

2. *After implementing, run Naïve Bayes with the multinomial model on the Amazon dataset (sentiment classification) and report results both for training and testing:*

```
import lxmls.classifiers.multinomial_naive_bayes as mnbb

mnb = mnbb.MultinomialNaiveBayes()
params_nb_sc = mnb.train(scr.train_X,scr.train_y)
y_pred_train = mnb.test(scr.train_X,params_nb_sc)
acc_train = mnb.evaluate(scr.train_y, y_pred_train)
y_pred_test = mnb.test(scr.test_X,params_nb_sc)
acc_test = mnb.evaluate(scr.test_y, y_pred_test)
print("Multinomial Naive Bayes Amazon Sentiment Accuracy train: %f test: %f"%(
    acc_train,acc_test))
```

3. *Observe that words that were not observed at training time cause problems at test time. Why? To solve this problem, apply a simple* add-one *smoothing technique: replace the expression in Eq. 1.9 for the estimation of the conditional probabilities by*

$$\hat{P}(w_j|c_k) = \frac{1 + \sum_{m \in \mathcal{I}_k} n_j(x^m)}{J + \sum_{i=1}^{J} \sum_{m \in \mathcal{I}_k} n_i(x^m)}.$$

where J is the number of distinct words.

This is a widely used smoothing strategy which has a Bayesian interpretation: it corresponds to choosing a uniform prior for the word distribution on both classes, and to replace the maximum likelihood criterion by a maximum a posteriori *approach. This is a form of* regularization, *preventing the model from* overfitting *on the training data. See* e.g. *Manning and Schütze (1999); Manning et al. (2008) for more information. Report the new accuracies.*

1.4 Discriminative Classifiers

In the previous sections we discussed generative classifiers, which require us to model the class prior and class conditional distributions ($P(Y)$ and $P(X|Y)$, respectively). Recall, however, that a classifier is *any* function which maps objects $x \in \mathcal{X}$ onto classes $y \in \mathcal{Y}$. While it's often useful to model how the data was generated, it's not required. Classifiers that do not model these distributions are called *discriminative* classifiers.

1.4.1 Features

For the purpose of understanding discriminative classifiers, it is useful to think about each $x \in \mathcal{X}$ as an abstract object which is subject to a set of descriptions or measurements, which are called *features*. A feature is simply a real number that describes the value of some property of x. For example, in the previous section, the features of a document were the number of times each word w_j appeared in it.

Let $g_1(x), \ldots, g_J(x)$ be J features of x. We call the vector

$$\boldsymbol{g}(x) = (g_1(x), \ldots, g_J(x)) \tag{1.10}$$

a *feature vector representation* of x. The map $\boldsymbol{g} : \mathcal{X} \to \mathbb{R}^J$ is called a *feature mapping*.

In NLP applications, features are often binary-valued and result from evaluating propositions such as:

$$g_1(x) \triangleq \begin{cases} 1, & \text{if sentence } x \text{ contains the word } \textit{Ronaldo} \\ 0, & \text{otherwise.} \end{cases} \tag{1.11}$$

$$g_2(x) \triangleq \begin{cases} 1, & \text{if all words in sentence } x \text{ are capitalized} \\ 0, & \text{otherwise.} \end{cases} \tag{1.12}$$

$$g_3(x) \triangleq \begin{cases} 1, & \text{if } x \text{ contains any of the words } \textit{amazing}, \textit{excellent} \text{ or } \textit{:-)} \\ 0, & \text{otherwise.} \end{cases} \tag{1.13}$$

In this example, the feature vector representation of the sentence ”Ronaldo shoots and scores an amazing goal!” would be $\boldsymbol{g}(x) = (1, 0, 1)$.

In multi-class learning problems, rather than associating features only with the input objects, it is useful to consider *joint feature mappings* $\boldsymbol{f} : \mathcal{X} \times \mathcal{Y} \to \mathbb{R}^D$. In that case, the *joint feature vector* $\boldsymbol{f}(x, y)$ can be seen as a collection of joint input-output measurements. For example:

$$f_1(x,y) \triangleq \begin{cases} 1, & \text{if } x \text{ contains } \textit{Ronaldo}, \text{ and topic } y \text{ is } \texttt{sport} \\ 0, & \text{otherwise.} \end{cases} \tag{1.14}$$

$$f_2(x,y) \triangleq \begin{cases} 1, & \text{if } x \text{ contains } \textit{Ronaldo}, \text{ and topic } y \text{ is } \texttt{politics} \\ 0, & \text{otherwise.} \end{cases} \tag{1.15}$$

A very simple form of defining a joint feature mapping which is often employed is via:

$$\begin{aligned} \boldsymbol{f}(x,y) &\triangleq \boldsymbol{g}(x) \otimes \boldsymbol{e}_y \\ &= (0, \ldots, 0, \underbrace{\boldsymbol{g}(x)}_{y\text{th slot}}, 0, \ldots, 0) \end{aligned} \tag{1.16}$$

where $\boldsymbol{g}(x) \in \mathbb{R}^J$ is a input feature vector, $\otimes$ is the Kronecker product ($[\boldsymbol{a} \otimes \boldsymbol{b}]_{ij} = a_i b_j$) and $\boldsymbol{e}_y \in \mathbb{R}^K$, with $[\boldsymbol{e}_y]_c = 1$ iff $y = c$, and 0 otherwise. Hence $\boldsymbol{f}(x,y) \in \mathbb{R}^{J \times K}$.

1.4.2 Inference

Linear classifiers are very popular in natural language processing applications. They make their decision based on the rule:

$$\hat{y} = \arg\max_{y\in\mathcal{Y}} \boldsymbol{w} \cdot \boldsymbol{f}(x,y). \tag{1.17}$$

where

- $\boldsymbol{w} \in \mathbb{R}^D$ is a *weight vector;*
- $\boldsymbol{f}(x,y) \in \mathbb{R}^D$ is a *feature vector;*
- $\boldsymbol{w} \cdot \boldsymbol{f}(x,y) = \sum_{d=1}^{D} w_d f_d(x,y)$ is the inner product between $\boldsymbol{w}$ and $\boldsymbol{f}(x,y)$.

Hence, each feature $f_d(x,y)$ has a weight w_d and, for each class $y \in \mathcal{Y}$, a score is computed by linearly combining all the weighted features. All these scores are compared, and a prediction is made by choosing the class with the largest score.

Remark 1.1 *With the design above (Eq. 1.16), and decomposing the weight vector as $\boldsymbol{w} = (\boldsymbol{w}_{c_1}, \ldots, \boldsymbol{w}_{c_K})$, we have that*

$$\boldsymbol{w} \cdot \boldsymbol{f}(x,y) = \boldsymbol{w}_y \cdot \boldsymbol{g}(x). \tag{1.18}$$

In words: each class $y \in \mathcal{Y}$ gets its own weight vector $\boldsymbol{w}_y$, and one defines a input feature vector $\boldsymbol{g}(x)$ that only looks at the input $x \in \mathcal{X}$. This representation is very useful when features only depend on input x since it allows a more compact representation. Note that the number of features is normally very large.

Remark 1.2 *The multinomial naïve Bayes classifier described in the previous section is an instance of a linear classifier. Recall that the naïve Bayes classifier predicts according to $\hat{y} = \arg\max_{y\in\mathcal{Y}} \hat{P}(y)\hat{P}(x|y)$. Taking logs, in the multinomial model for document classification this is equivalent to:*

$$\begin{aligned}
\hat{y} &= \arg\max_{y\in\mathcal{Y}} \log \hat{P}(y) + \log \hat{P}(x|y) \\
&= \arg\max_{y\in\mathcal{Y}} \log \hat{P}(y) + \sum_{j=1}^{J} n_j(x) \log \hat{P}(w_j|y) \\
&= \arg\max_{y\in\mathcal{Y}} \boldsymbol{w}_y \cdot \boldsymbol{g}(x), \qquad (1.19)
\end{aligned}$$

where

$$\begin{aligned}
\boldsymbol{w}_y &= \left(b_y, \log \hat{P}(w_1|y), \ldots, \log \hat{P}(w_J|y)\right) \\
b_y &= \log \hat{P}(y) \\
\boldsymbol{g}(x) &= (1, n_1(x), \ldots, n_J(x)). \qquad (1.20)
\end{aligned}$$

Hence, the multinomial model yields a prediction rule of the form

$$\hat{y} = \arg\max_{y\in\mathcal{Y}} \boldsymbol{w}_y \cdot \boldsymbol{g}(x). \tag{1.21}$$

1.4.3 Online Discriminative Algorithms

We now discuss two discriminative classification algorithms. These two algorithms are called *online* (or *stochastic*) algorithms because they only process one data point (in our example, one document) at a time. Algorithms which look at the whole dataset at once are called *offline*, or *batch* algorithms, and will be discussed later.

Perceptron

The *perceptron* (Rosenblatt, 1958) is perhaps the oldest algorithm used to train a linear classifier. The perceptron works as follows: at each round, it takes an element x from the dataset, and uses the current model to make a prediction. If the prediction is correct, nothing happens. Otherwise, the model is corrected by adding the feature vector w.r.t. the correct output and subtracting the feature vector w.r.t. the predicted (wrong) output. Then, it proceeds to the next round.

Algorithm 2 Averaged perceptron

```
input: dataset $\mathcal{D}$, number of rounds $R$
initialize $t = 0, \boldsymbol{w}^t = \mathbf{0}$
for $r = 1$ to $R$ do
    $\mathcal{D}_s = \text{shuffle}(\mathcal{D})$
    for $i = 1$ to $M$ do
        $m = \mathcal{D}_s(i)$
        take training pair $(x^m, y^m)$ and predict using the current model:
            $$\hat{y} \leftarrow \arg\max_{y' \in \mathcal{Y}} \boldsymbol{w}^t \cdot \boldsymbol{f}(x^m, y')$$
        update the model: $\boldsymbol{w}^{t+1} \leftarrow \boldsymbol{w}^t + \boldsymbol{f}(x^m, y^m) - \boldsymbol{f}(x^m, \hat{y})$
        $t = t + 1$
    end for
end for
output: the averaged model $\hat{\boldsymbol{w}} \leftarrow \frac{1}{t} \sum_{i=1}^{t} \boldsymbol{w}^i$
```

Alg. 2 shows the pseudo-code of the perceptron algorithm. As it can be seen, it is remarkably simple; yet it often reaches a very good performance, often better than the Naïve Bayes, and usually not much worse than maximum entropy models or SVMs (which will be described in the following section). [2]

A weight vector $\boldsymbol{w}$ defines a *separating hyperplane* if it classifies all the training data correctly, *i.e.*, if $y^m = \arg\max_{y \in \mathcal{Y}} \boldsymbol{w} \cdot \boldsymbol{f}(x^m, y)$ hold for $m = 1, \ldots, M$. A dataset $\mathcal{D}$ is *separable* if such a weight vector exists (in general, $\boldsymbol{w}$ is not unique). A very important property of the perceptron algorithm is the following: if $\mathcal{D}$ is separable, then the number of mistakes made by the perceptron algorithm until it finds a separating hyperplane is *finite*. This means that if the data are separable, the perceptron will eventually find a separating hyperplane $\boldsymbol{w}$.

There are other variants of the perceptron (e.g., with regularization) which we omit for brevity.

Exercise 1.2 *We provide an implementation of the perceptron algorithm in the class* `Perceptron` *(file* `perceptron.py`*).*

1. *Run the following commands to generate a simple dataset similar to the one plotted on Figure 1.1:*

```
import lxmls.readers.simple_data_set as sds
sd = sds.SimpleDataSet(nr_examples=100, g1 = [[-1,-1],1], g2 = [[1,1],1], balance=0.
    5, split=[0.5,0,0.5])
```

2. *Run the perceptron algorithm on the simple dataset previously generated and report its train and test set accuracy:*

```
import lxmls.classifiers.perceptron as percc

perc = percc.Perceptron()
params_perc_sd = perc.train(sd.train_X,sd.train_y)
y_pred_train = perc.test(sd.train_X,params_perc_sd)
acc_train = perc.evaluate(sd.train_y, y_pred_train)
y_pred_test = perc.test(sd.test_X,params_perc_sd)
acc_test = perc.evaluate(sd.test_y, y_pred_test)
print("Perceptron Simple Dataset Accuracy train: %f test: %f"%(acc_train,acc_test))
```

3. *Plot the decision boundary found:*

```
fig,axis = sd.plot_data()
fig,axis = sd.add_line(fig,axis,params_perc_sd,"Perceptron","blue")
```

Change the code to save the intermediate weight vectors, and plot the decision boundaries every five iterations. What do you observe?

4. *Run the perceptron algorithm on the Amazon dataset.*

[2] Actually, we are showing a more robust variant of the perceptron, which averages the weight vector as a post-processing step.

Algorithm 3 MIRA

```
input: dataset $\mathcal{D}$, parameter $\lambda$, number of rounds $R$
initialize $t = 0, \boldsymbol{w}^t = \mathbf{0}$
for $r = 1$ to $R$ do
  $\mathcal{D}_s = \text{shuffle}(\mathcal{D})$
  for $i = 1$ to $M$ do
    $m = \mathcal{D}_s(i)$
    $t = t + 1$
    take training pair $(x^m, y^m)$ and predict using the current model:
        $\hat{y} \leftarrow \arg\max_{y' \in \mathcal{Y}} \boldsymbol{w}^t \cdot \boldsymbol{f}(x^m, y')$
    compute loss: $\ell^t = \boldsymbol{w}^t \cdot \boldsymbol{f}(x^m, \hat{y}) - \boldsymbol{w}^t \cdot \boldsymbol{f}(x^m, y^m) + \rho(\hat{y}, y^m)$
    compute stepsize: $\eta^t = \min\left\{\lambda^{-1}, \frac{\ell^t}{\|\boldsymbol{f}(x^m,y^m) - \boldsymbol{f}(x^m,\hat{y})\|^2}\right\}$
    update the model: $\boldsymbol{w}^{t+1} \leftarrow \boldsymbol{w}^t + \eta^t(\boldsymbol{f}(x^m, y^m) - \boldsymbol{f}(x^m, \hat{y}))$
  end for
end for
output: the averaged model $\hat{\boldsymbol{w}} \leftarrow \frac{1}{t}\sum_{i=1}^{t} \boldsymbol{w}^i$
```

Margin Infused Relaxed Algorithm (MIRA)

The MIRA algorithm (Crammer and Singer, 2002; Crammer et al., 2006) has achieved very good performance in NLP problems. Recall that the Perceptron takes an input pattern and, if its prediction is wrong, adds the quantity $[\boldsymbol{f}(x^m, y^m) - \boldsymbol{f}(x^m, \hat{y})]$ to the weight vector. MIRA changes this by adding $\eta^t[\boldsymbol{f}(x^m, y^m) - \boldsymbol{f}(x^m, \hat{y})]$ to the weight vector. The difference is the step size η^t, which depends on the iteration t.

There is a theoretical basis for this algorithm, which we now briefly explain. At each round t, MIRA updates the weight vector by solving the following optimization problem:

$$\boldsymbol{w}^{t+1} \leftarrow \arg\min_{\boldsymbol{w},\xi} \quad \xi + \frac{\lambda}{2}\|\boldsymbol{w} - \boldsymbol{w}^t\|^2 \tag{1.22}$$

$$\text{s.t.} \quad \boldsymbol{w} \cdot \boldsymbol{f}(x^m, y^m) \geq \boldsymbol{w} \cdot \boldsymbol{f}(x^m, \hat{y}) + 1 - \xi \tag{1.23}$$

$$\xi \geq 0, \tag{1.24}$$

where $\hat{y} = \arg\max_{y' \in \mathcal{Y}} \boldsymbol{w}^t \cdot \boldsymbol{f}(x^m, y')$ is the prediction using the model with weight vector $\boldsymbol{w}^t$. By inspecting Eq. 1.22 we see that MIRA attempts to achieve a tradeoff between *conservativeness* (penalizing large changes from the previous weight vector via the term $\frac{\lambda}{2}\|\boldsymbol{w} - \boldsymbol{w}^t\|^2$) and *correctness* (by requiring, through the constraints, that the new model $\boldsymbol{w}^{t+1}$ "separates" the true output from the prediction with a margin (although slack $\xi \geq 0$ is allowed).[3] Note that, if the prediction is correct ($\hat{y} = y^m$) the solution of the problem Eq. 1.22 leaves the weight vector unchanged ($\boldsymbol{w}^{t+1} = \boldsymbol{w}^t$). This quadratic programming problem has a closed form solution:[4]

$$\boldsymbol{w}^{t+1} \leftarrow \boldsymbol{w}^t + \eta^t(\boldsymbol{f}(x^m, y^m) - \boldsymbol{f}(x^m, \hat{y})),$$

with

$$\eta^t = \min\left\{\lambda^{-1}, \frac{\boldsymbol{w}^t \cdot \boldsymbol{f}(x^m, \hat{y}) - \boldsymbol{w}^t \cdot \boldsymbol{f}(x^m, y^m) + \rho(\hat{y}, y^m)}{\|\boldsymbol{f}(x^m, y^m) - \boldsymbol{f}(x^m, \hat{y})\|^2}\right\},$$

where $\rho : \mathcal{Y} \times \mathcal{Y} \to \mathbb{R}_+$ is a non-negative cost function, such that $\rho(\hat{y}, y)$ is the cost incurred by predicting $\hat{y}$ when the true output is y; we assume $\rho(y, y) = 0$ for all $y \in \mathcal{Y}$. For simplicity, we focus here on the 0/1-cost (but keep in mind that other cost functions are possible):

$$\rho(\hat{y}, y) = \begin{cases} 1 & \text{if } \hat{y} \neq y \\ 0 & \text{otherwise.} \end{cases} \tag{1.25}$$

MIRA is depicted in Alg. 3. For other variants of MIRA, see Crammer et al. (2006).

Exercise 1.3 *We provide an implementation of the MIRA algorithm. Compare it with the perceptron for various values*

[3] The intuition for this large margin separation is the same for support vector machines, which will be discussed in §1.4.4.

[4] Note that the perceptron updates are identical, except that we always have $\eta_t = 1$.

of λ

```
import lxmls.classifiers.mira as mirac

mira = mirac.Mira()
mira.regularizer = 1.0 # This is lambda
params_mira_sd = mira.train(sd.train_X,sd.train_y)
y_pred_train = mira.test(sd.train_X,params_mira_sd)
acc_train = mira.evaluate(sd.train_y, y_pred_train)
y_pred_test = mira.test(sd.test_X,params_mira_sd)
acc_test = mira.evaluate(sd.test_y, y_pred_test)
print("Mira Simple Dataset Accuracy train: %f test: %f"%(acc_train,acc_test))
fig,axis = sd.add_line(fig,axis,params_mira_sd,"Mira","green")

params_mira_sc = mira.train(scr.train_X,scr.train_y)
y_pred_train = mira.test(scr.train_X,params_mira_sc)
acc_train = mira.evaluate(scr.train_y, y_pred_train)
y_pred_test = mira.test(scr.test_X,params_mira_sc)
acc_test = mira.evaluate(scr.test_y, y_pred_test)
print("Mira Amazon Sentiment Accuracy train: %f test: %f"%(acc_train,acc_test))
```

Compare the results achieved and separating hyperplanes found.

1.4.4 Batch Discriminative Classifiers

As we have mentioned, the perceptron and MIRA algorithms are called *online* or *stochastic* because they look at one data point at a time. We now describe two discriminative classifiers which look at all points at once; these are called *offline* or *batch* algorithms.

Maximum Entropy Classifiers

The notion of *entropy* in the context of Information Theory (Shannon, 1948) is one of the most significant advances in mathematics in the twentieth century. The principle of *maximum entropy* (which appears under different names, such as “maximum mutual information” or “minimum Kullback-Leibler divergence”) plays a fundamental role in many methods in statistics and machine learning (Jaynes, 1982). [5] The basic rationale is that choosing the model with the highest entropy (subject to constraints that depend on the observed data) corresponds to making the fewest possible assumptions regarding what was unobserved, making uncertainty about the model as large as possible.

For example, if we throw a die and want to estimate the probability of its outcomes, the distribution with the highest entropy would be the uniform distribution (each outcome having of probability a $1/6$). Now suppose that we are only told that outcomes $\{1,2,3\}$ occurred 10 times in total, and $\{4,5,6\}$ occurred 30 times in total, then the principle of maximum entropy would lead us to estimate $P(1) = P(2) = P(3) = 1/12$ and $P(4) = P(5) = P(6) = 1/4$ (i.e., outcomes would be uniform within each of the two groups).[6]

This example could be presented in a more formal way. Suppose that we want to use binary features to represent the outcome of the die throw. We use two features: $f_{123}(x,y) = 1$ if and only if $y \in \{1,2,3\}$, and $f_{456}(x,y) = 1$ if and only if $y \in \{4,5,6\}$. Our observations state that in 40 throws, we observed f_{123} 10 times (25%) and f_{456} 30 times (75%). The maximum entropy principle states that we want to find the parameters $\boldsymbol{w}$ of our model, and consequently the probability distribution $P_{\boldsymbol{w}}(Y|X)$, which makes f_{123} have an expected value of 0.25 and f_{456} have an expected value of 0.75. These constraints, $E[f_{123}] = 0.25$ and $E[f_{456}] = 0.75$, are known as *first moment matching constraints*.[7]

An important fundamental result, which we will not prove here, is that the maximum entropy distribution $P_{\boldsymbol{w}}(Y|X)$ under first moment matching constraints is a *log-linear model*. [8] It has the following parametric form:

$$P_{\boldsymbol{w}}(y|x) = \frac{\exp(\boldsymbol{w} \cdot \boldsymbol{f}(x,y))}{Z(\boldsymbol{w},x)} \tag{1.26}$$

[5] For an excellent textbook on Information Theory, we recommend Cover et al. (1991).

[6] For an introduction of maximum entropy models, along with pointers to the literature, see `http://www.cs.cmu.edu/~aberger/maxent.html`.

[7] In general, these constraints mean that feature expectations under that distribution $\frac{1}{M}\sum_m E_{Y\sim P_{\boldsymbol{w}}}[\boldsymbol{f}(x_m,Y)]$ must match the observed relative frequencies $\frac{1}{M}\sum_m \boldsymbol{f}(x_m,y_m)$.

[8] Also called a a Boltzmann distribution, or an exponential family of distributions.

The denominator in Eq. 1.26 is called the *partition function*:

$$Z(\boldsymbol{w}, x) = \sum_{y' \in \mathcal{Y}} \exp(\boldsymbol{w} \cdot \boldsymbol{f}(x, y')). \tag{1.27}$$

An important property of the partition function is that the gradient of its logarithm equals the feature expectations:

$$\begin{aligned} \nabla_{\boldsymbol{w}} \log Z(\boldsymbol{w}, x) &= E_{\boldsymbol{w}}[\boldsymbol{f}(x, Y)] \\ &= \sum_{y' \in \mathcal{Y}} P_{\boldsymbol{w}}(y'|x) \boldsymbol{f}(x, y'). \end{aligned} \tag{1.28}$$

The average conditional log-likelihood is:

$$\begin{aligned} \mathcal{L}(\boldsymbol{w}; \mathcal{D}) &= \frac{1}{M} \log P_{\boldsymbol{w}}(y^1, \ldots, y^M | x^1, \ldots, x^M) \\ &= \frac{1}{M} \log \prod_{m=1}^{M} P_{\boldsymbol{w}}(y^m | x^m) \\ &= \frac{1}{M} \sum_{m=1}^{M} \log P_{\boldsymbol{w}}(y^m | x^m) \\ &= \frac{1}{M} \sum_{m=1}^{M} \left(\boldsymbol{w} \cdot \boldsymbol{f}(x^m, y^m) - \log Z(\boldsymbol{w}, x^m) \right). \end{aligned} \tag{1.29}$$

We try to find the parameters $\boldsymbol{w}$ that maximize the log-likelihood $\mathcal{L}(\boldsymbol{w}; \mathcal{D})$; to avoid overfitting, we add a regularization term that penalizes values of $\boldsymbol{w}$ that have a high magnitude. The optimization problem becomes:

$$\begin{aligned} \hat{\boldsymbol{w}} &= \arg\max_{\boldsymbol{w}} \mathcal{L}(\boldsymbol{w}; \mathcal{D}) - \frac{\lambda}{2} \|\boldsymbol{w}\|^2 \\ &= \arg\min_{\boldsymbol{w}} -\mathcal{L}(\boldsymbol{w}; \mathcal{D}) + \frac{\lambda}{2} \|\boldsymbol{w}\|^2. \end{aligned} \tag{1.30}$$

Here we use the squared L_2-norm as the regularizer,[9] but other norms are possible. The scalar $\lambda \geq 0$ controls the amount of regularization. Unlike the naïve Bayes examples, this optimization problem does not have a closed form solution in general; hence we need to resort to numerical optimization (see section 0.7). Let $F_\lambda(\boldsymbol{w}; \mathcal{D}) = -\mathcal{L}(\boldsymbol{w}; \mathcal{D}) + \frac{\lambda}{2}\|\boldsymbol{w}\|^2$ be the objective function in Eq. 1.30. This function is convex, which implies that a local optimum of Eq. 1.30 is also a global optimum. $F_\lambda(\boldsymbol{w}; \mathcal{D})$ is also differentiable: its gradient is

$$\begin{aligned} \nabla_{\boldsymbol{w}} F_\lambda(\boldsymbol{w}; \mathcal{D}) &= \frac{1}{M} \sum_{m=1}^{M} (-\boldsymbol{f}(x^m, y^m) + \nabla_{\boldsymbol{w}} \log Z(\boldsymbol{w}, x^m)) + \lambda \boldsymbol{w} \\ &= \frac{1}{M} \sum_{m=1}^{M} (-\boldsymbol{f}(x^m, y^m) + E_{\boldsymbol{w}}[\boldsymbol{f}(x^m, Y)]) + \lambda \boldsymbol{w}. \end{aligned} \tag{1.31}$$

A batch gradient method to optimize Eq. 1.30 is shown in Alg. 4. Essentially, Alg. 4 iterates through the following updates until convergence:

$$\begin{aligned} \boldsymbol{w}^{t+1} &\leftarrow \boldsymbol{w}^t - \eta_t \nabla_{\boldsymbol{w}} F_\lambda(\boldsymbol{w}^t; \mathcal{D}) \\ &= (1 - \lambda \eta_t) \boldsymbol{w}^t + \eta_t \frac{1}{M} \sum_{m=1}^{M} \left(\boldsymbol{f}(x^m, y^m) - E_{\boldsymbol{w}}[\boldsymbol{f}(x^m, Y)] \right). \end{aligned} \tag{1.32}$$

Convergence is ensured for suitable stepsizes η_t. Monotonic decrease of the objective value can also be ensured if η_t is chosen with a suitable line search method, such as Armijo's rule (Nocedal and Wright, 1999). In practice, more sophisticated methods exist for optimizing Eq. 1.30, such as conjugate gradient or L-BFGS. The latter is an example of a quasi-Newton method, which only requires gradient information, but uses past gradients to try to construct second order (Hessian) approximations.

In large-scale problems (very large M) batch methods are slow. *Online* or *stochastic* optimization are at-

[9] In a Bayesian perspective, this corresponds to choosing independent Gaussian priors $p(w_d) \sim \mathcal{N}(0; 1/\lambda^2)$ for each dimension of the weight vector.

Algorithm 4 Batch Gradient Descent for Maximum Entropy

```
input: $\mathcal{D}$, $\lambda$, number of rounds $T$,
learning rate sequence $(\eta_t)_{t=1,...,T}$
initialize $\boldsymbol{w}^1 = \mathbf{0}$
for $t = 1$ to $T$ do
    for $m = 1$ to $M$ do
        take training pair $(x^m, y^m)$ and compute conditional probabilities using the current model, for each $y' \in \mathcal{Y}$:
            $$P_{\boldsymbol{w}^t}(y'|x^m) = \frac{\exp(\boldsymbol{w}^t \cdot \boldsymbol{f}(x^m, y'))}{Z(\boldsymbol{w}, x^m)}$$
        compute the feature vector expectation:
            $$E_{\boldsymbol{w}}[\boldsymbol{f}(x^m, Y)] = \sum_{y' \in \mathcal{Y}} P_{\boldsymbol{w}^t}(y'|x^m)\boldsymbol{f}(x^m, y')$$
    end for
    choose the stepsize $\eta_t$ using, e.g., Armijo's rule
    update the model:
        $$\boldsymbol{w}^{t+1} \leftarrow (1 - \lambda\eta_t)\boldsymbol{w}^t + \eta_t M^{-1} \sum_{m=1}^{M} (\boldsymbol{f}(x^m, y^m) - E_{\boldsymbol{w}}[\boldsymbol{f}(x^m, Y)])$$
end for
output: $\hat{\boldsymbol{w}} \leftarrow \boldsymbol{w}^{T+1}$
```

tractive alternative methods. Stochastic gradient methods make "noisy" gradient updates by considering only a single instance at the time. The resulting algorithm, called Stochastic Gradient Descent (SGD) is shown as Alg. 5. At each round t, an instance $m(t)$ is chosen, either randomly (stochastic variant) or by cycling through the dataset (online variant). The stepsize sequence must decrease with t: typically, $\eta_t = \eta_0 t^{-\alpha}$ for some $\eta_0 > 0$ and $\alpha \in [1, 2]$, tuned in a development partition or with cross-validation.

Exercise 1.4 *We provide an implementation of the L-BFGS algorithm for training maximum entropy models in the class* `MaxEnt_batch`*, as well as an implementation of the SGD algorithm in the class* `MaxEnt_online`*.*

1. *Train a maximum entropy model using L-BFGS on the Simple data set (try different values of λ). Compare the results with the previous methods. Plot the decision boundary.*

```
import lxmls.classifiers.max_ent_batch as mebc

me_lbfgs = mebc.MaxEntBatch()
me_lbfgs.regularizer = 1.0
params_meb_sd = me_lbfgs.train(sd.train_X,sd.train_y)
y_pred_train = me_lbfgs.test(sd.train_X,params_meb_sd)
acc_train = me_lbfgs.evaluate(sd.train_y, y_pred_train)
y_pred_test = me_lbfgs.test(sd.test_X,params_meb_sd)
acc_test = me_lbfgs.evaluate(sd.test_y, y_pred_test)
print("Max-Ent batch Simple Dataset Accuracy train: %f test: %f"%(acc_train,acc_test
    ))

fig,axis = sd.add_line(fig,axis,params_meb_sd,"Max-Ent-Batch","orange")
```

2. *Train a maximum entropy model using L-BFGS, on the Amazon dataset (try different values of λ) and report training and test set accuracy. What do you observe?*

```
params_meb_sc = me_lbfgs.train(scr.train_X,scr.train_y)
y_pred_train = me_lbfgs.test(scr.train_X,params_meb_sc)
acc_train = me_lbfgs.evaluate(scr.train_y, y_pred_train)
y_pred_test = me_lbfgs.test(scr.test_X,params_meb_sc)
acc_test = me_lbfgs.evaluate(scr.test_y, y_pred_test)
```

Algorithm 5 SGD for Maximum Entropy

```
input: $\mathcal{D}$, $\lambda$, number of rounds $T$,
learning rate sequence $(\eta_t)_{t=1,\ldots,T}$
initialize $\boldsymbol{w}^1 = \mathbf{0}$
for $t = 1$ to $T$ do
    choose $m = m(t)$ randomly
    take training pair $(x^m, y^m)$ and compute conditional probabilities using the current model, for each $y' \in \mathcal{Y}$:
        $$P_{\boldsymbol{w}^t}(y'|x^m) = \frac{\exp(\boldsymbol{w}^t \cdot \boldsymbol{f}(x^m, y'))}{Z(\boldsymbol{w}, x^m)}$$
    compute the feature vector expectation:
        $$E_{\boldsymbol{w}}[\boldsymbol{f}(x^m, Y)] = \sum_{y' \in \mathcal{Y}} P_{\boldsymbol{w}^t}(y'|x^m)\boldsymbol{f}(x^m, y')$$
    update the model:
        $$\boldsymbol{w}^{t+1} \leftarrow (1 - \lambda\eta_t)\boldsymbol{w}^t + \eta_t\left(\boldsymbol{f}(x^m, y^m) - E_{\boldsymbol{w}}[\boldsymbol{f}(x^m, Y)]\right)$$
end for
output: $\hat{\boldsymbol{w}} \leftarrow \boldsymbol{w}^{T+1}$
```

```python
print("Max-Ent Batch Amazon Sentiment Accuracy train: %f test: %f"%(acc_train,
    acc_test))
```

3. *Now, fix* $\lambda = 1.0$ *and train with SGD (you might try to adjust the initial step). Compare the objective values obtained during training with those obtained with L-BFGS. What do you observe?*

```python
import lxmls.classifiers.max_ent_online as meoc

me_sgd = meoc.MaxEntOnline()
me_sgd.regularizer = 1.0
params_meo_sc = me_sgd.train(scr.train_X,scr.train_y)
y_pred_train = me_sgd.test(scr.train_X,params_meo_sc)
acc_train = me_sgd.evaluate(scr.train_y, y_pred_train)
y_pred_test = me_sgd.test(scr.test_X,params_meo_sc)
acc_test = me_sgd.evaluate(scr.test_y, y_pred_test)
print("Max-Ent Online Amazon Sentiment Accuracy train: %f test: %f"%(acc_train,
    acc_test))
```

Support Vector Machines

Support vector machines are also a discriminative approach, but they are not a probabilistic model at all. The basic idea is that, if the goal is to accurately predict outputs (according to some cost function), we should focus on that goal in the first place, rather than trying to estimate a probability distribution ($P(Y|X)$ or $P(X, Y)$), which is a more difficult problem. As Vapnik (1995) puts it, "do not solve an estimation problem of interest by solving a more general (harder) problem as an intermediate step."

We next describe the *primal* problem associated with multi-class support vector machines (Crammer and Singer, 2002), which is of primary interest in natural language processing. There is a significant amount of literature about Kernel Methods (Schölkopf and Smola, 2002; Shawe-Taylor and Cristianini, 2004) mostly focused on the *dual* formulation. We will not discuss non-linear kernels or this dual formulation here.[10]

Consider $\rho(y', y)$ as a non-negative cost function, representing the cost of assigning a label y' when the correct label was y. For simplicity, we focus here on the $0/1$-cost defined by Equation 1.25 (but keep in mind

[10] The main reason why we prefer to discuss the primal formulation with linear kernels is that the resulting algorithms run in linear time (or less), while known kernel-based methods are quadratic with respect to M. In large-scale problems (large M) the former are thus more appealing.

Algorithm 6 Stochastic Subgradient Descent for SVMs

```
input: $\mathcal{D}$, $\lambda$, number of rounds $T$,
    learning rate sequence $(\eta_t)_{t=1,\ldots,T}$
initialize $\boldsymbol{w}^1 = \mathbf{0}$
for $t = 1$ to $T$ do
    choose $m = m(t)$ randomly
    take training pair $(x^m, y^m)$ and compute the "cost-augmented prediction" under the current model:
        $\tilde{y} = \arg\max_{y' \in \mathcal{Y}} \boldsymbol{w}^t \cdot \boldsymbol{f}(x^m, y') - \boldsymbol{w}^t \cdot \boldsymbol{f}(x^m, y^m) + \rho(y', y)$
    update the model:
        $\boldsymbol{w}^{t+1} \leftarrow (1 - \lambda\eta_t)\boldsymbol{w}^t + \eta_t \left(\boldsymbol{f}(x^m, y^m) - \boldsymbol{f}(x^m, \tilde{y})\right)$
end for
output: $\hat{\boldsymbol{w}} \leftarrow \boldsymbol{w}^{T+1}$
```

that other cost functions are possible). The *hinge loss*[11] is the function

$$\ell(\boldsymbol{w}; x, y) = \max_{y' \in \mathcal{Y}} \left[\boldsymbol{w} \cdot \boldsymbol{f}(x, y') - \boldsymbol{w} \cdot \boldsymbol{f}(x, y) + \rho(y', y)\right]. \tag{1.33}$$

Note that the objective of Eq. 1.33 becomes zero when $y' = y$. Hence, we always have $\ell(\boldsymbol{w}; x, y) \geq 0$. Moreover, if ρ is the $0/1$ cost, we have $\ell(\boldsymbol{w}; x, y) = 0$ if and only if the weight vector is such that the model makes a correct prediction with a *margin* greater than 1: *i.e.*, $\boldsymbol{w} \cdot \boldsymbol{f}(x, y) \geq \boldsymbol{w} \cdot \boldsymbol{f}(x, y') + 1$ for all $y' \neq y$. Otherwise, a positive loss is incurred. The idea behind this formulation is that not only do we want to make a correct prediction, but we want to make a *confident* prediction; this is why we have a loss unless the prediction is correct with some margin.

Support vector machines (SVM) tackle the following optimization problem:

$$\hat{\boldsymbol{w}} \quad = \quad \arg\min_{\boldsymbol{w}} \sum_{m=1}^{M} \ell(\boldsymbol{w}; x^m, y^m) + \frac{\lambda}{2}\|\boldsymbol{w}\|^2, \tag{1.34}$$

where we also use the squared L_2-norm as the regularizer. For the $0/1$-cost, the problem in Eq. 1.34 is equivalent to:

$$\arg\min_{\boldsymbol{w}, \xi} \quad \textstyle\sum_{m=1}^{M} \xi_m + \frac{\lambda}{2}\|\boldsymbol{w}\|^2 \tag{1.35}$$

$$\text{s.t.} \quad \boldsymbol{w} \cdot \boldsymbol{f}(x^m, y^m) \geq \boldsymbol{w} \cdot \boldsymbol{f}(x^m, \tilde{y}^m) + 1 - \xi_m, \quad \forall m, \tilde{y}^m \in \mathcal{Y} \setminus \{y^m\}. \tag{1.36}$$

Geometrically, we are trying to choose the linear classifier that yields the largest possible separation margin, while we allow some violations, penalizing the amount of slack via extra variables $\xi_1, \ldots, \xi_m$. There is now a trade-off: increasing the slack variables ξ_m makes it easier to satisfy the constraints, but it will also increase the value of the cost function.

Problem 1.34 does not have a closed form solution. Moreover, unlike maximum entropy models, the objective function in 1.34 is non-differentiable, hence smooth optimization is not possible. However, it is still convex, which ensures that any local optimum is the global optimum. Despite not being differentiable, we can still define a *subgradient* of the objective function (which generalizes the concept of gradient), which enables us to apply subgradient-based methods. A stochastic subgradient algorithm for solving Eq. 1.34 is illustrated as Alg. 6. The similarity with maximum entropy models (Alg. 5) is striking: the only difference is that, instead of computing the feature vector expectation using the current model, we compute the feature vector associated with the cost-augmented prediction using the current model.

A variant of this algorithm was proposed by Shalev-Shwartz et al. (2007) under the name *Pegasos*, with excellent properties in large-scale settings. Other algorithms and software packages for training SVMs that have become popular are SVMLight (`http://svmlight.joachims.org`) and LIBSVM (`http://www.csie.ntu.edu.tw/~cjlin/libsvm/`), which allow non-linear kernels. These will generally be more suitable for smaller datasets, where high accuracy optimization can be obtained without much computational effort.

[11] The hinge loss for the $0/1$ cost is sometimes defined as $\ell(\boldsymbol{w}; x, y) = \max\{0, \max_{y' \neq y} \boldsymbol{w} \cdot \boldsymbol{f}(x, y') - \boldsymbol{w} \cdot \boldsymbol{f}(x, y) + 1\}$. Given our definition of $\rho(\hat{y}, y)$, note that the two definitons are equivalent.

Remark 1.3 *Note the similarity between the stochastic (sub-)gradient algorithms (Algs. 5–6) and the online algorithms seen above (perceptron and MIRA).*

Exercise 1.5 *Run the SVM primal algorithm. Then, repeat the MaxEnt exercise now using SVMs, for several values of λ:*

```
import lxmls.classifiers.svm as svmc

svm = svmc.SVM()
svm.regularizer = 1.0 # This is lambda
params_svm_sd = svm.train(sd.train_X,sd.train_y)
y_pred_train = svm.test(sd.train_X,params_svm_sd)
acc_train = svm.evaluate(sd.train_y, y_pred_train)
y_pred_test = svm.test(sd.test_X,params_svm_sd)
acc_test = svm.evaluate(sd.test_y, y_pred_test)
print("SVM Online Simple Dataset Accuracy train: %f test: %f"%(acc_train,acc_test))

fig,axis = sd.add_line(fig,axis,params_svm_sd,"SVM","orange")

params_svm_sc = svm.train(scr.train_X,scr.train_y)
y_pred_train = svm.test(scr.train_X,params_svm_sc)
acc_train = svm.evaluate(scr.train_y, y_pred_train)
y_pred_test = svm.test(scr.test_X,params_svm_sc)
acc_test = svm.evaluate(scr.test_y, y_pred_test)
print("SVM Online Amazon Sentiment Accuracy train: %f test: %f"%(acc_train,acc_test))
```

Compare the results achieved and separating hyperplanes found.

1.5 Comparison

Table 1.1 provides a high-level comparison among the different algorithms discussed in this chapter.

	Naive Bayes	Perceptron	MIRA	MaxEnt	SVMs
Generative/Discriminative	G	D	D	D	D
Performance if true model not in the hypothesis class	Bad	Fair (may not converge)	Good	Good	Good
Performance if features overlap	Fair	Good	Good	Good	Good
Training	Closed Form	Easy	Easy	Fair	Fair
Hyperparameters to tune	1 (smoothing)	0	1	1	1

Table 1.1: Comparison among different algorithms.

Exercise 1.6

- *Using the simple dataset run the different algorithms varying some characteristics of the data: like the number of points, variance (hence separability), class balance. Use function* run_all_classifiers *in file* labs/run_all_classifiers.py *which receives a dataset and plots all decisions boundaries and accuracies. What can you say about the methods when the amount of data increases? What about when the classes become too unbalanced?*

1.6 Final remarks

Some implementations of the discussed algorithms are available on the Web:

- SVMLight: `http://svmlight.joachims.org`
- LIBSVM: `http://www.csie.ntu.edu.tw/~cjlin/libsvm/`
- Maximum Entropy: `http://homepages.inf.ed.ac.uk/lzhang10/maxent_toolkit.html`
- MALLET: `http://mallet.cs.umass.edu/`.

Day 2

Non-Linear Classifiers

Today's class will introduce modern neural network models, commonly known as deep learning models. We will learn the concept of *computation graph*, a general way of describing complex functions as composition of simpler functions. We will also learn about *Backpropagation*, a generic solution for gradient-descent based optimization in computation graphs.

2.1 Today's assignment

Your objective today should be to understand fully the concept of Backpropagation. For this, we will code Backpropagation in Numpy on a simple feed forward network. Then we will learn about the Pytorch module, which allows to easily create dynamic computation graphs and computes Backpropagation automatically for us. If you are new to the topic, you should aim to understand the concept of computation graph, finish the Backpropagation exercise and attain a basic understanding of Pytorch. If you already know Backpropagation well and have experience with normal Python, you should aim to complete the whole day.

2.2 Introduction to Deep Learning and Pytorch

Deep learning is the name behind the latest wave of neural network research. This is a very old topic, dating from the first half of the 20th century, that has attained formidable impact in the machine learning community recently. There is nothing particularly difficult in deep learning. You have already visited all the mathematical principles you need in the first days of the labs of this school. At their core, deep learning models are just functions mapping vector inputs $\mathbf{x}$ to vector outputs $\mathbf{y}$, constructed by composing linear and non-linear functions. This composition can be expressed in the form of a *computation graph*, where each node applies a function to its inputs and passes the result as its output. The parameters of the model are the weights given to the different inputs of nodes. This architecture vaguely resembles synapse strengths in human neural networks, hence the name artificial neural networks.

Since neural networks are just compositions of simple functions, we can apply the chain rule to derive gradients and learn the parameters of neural networks regardless of their complexity. See Section 0.7.3 for a refresh on the basic concept. We will also refer to the gradient learning methods introduced in Section 1.4.4. Today we will focus on *feed-forward networks*. The topic of *recurrent neural networks* (RNNs) will be visited in a posterior chapter.

Some of the changes that led to the surge of deep learning are not only improvements on the existing neural network algorithms, but also the increase in the amount of data available and computing power. In particular, the use of Graphical Processing Units (GPUs) has allowed neural networks to be applied to very large datasets. Working with GPUs is not trivial as it requires dealing with specialized hardware. Luckily, as it is often the case, we are one Python import away from solving this problem.

For the particular case of deep learning, there is a growing number of toolboxes with python bindings that allow you to design custom computational graphs for GPUs some of the best known are Theano[1], TensorFlow[2] and Pytorch[3].

[1] http://deeplearning.net/software/theano/

[2] https://www.tensorflow.org/

[3] http://pytorch.org/

In these labs we will be working with Pytorch. Pytorch allows us to create computation graphs of arbitrary complexity and automatically compute the gradients of the cost with respect to any parameter. It will also produce CUDA-compatible code for use with GPUs. One salient property of Pytorch, shared with other toolkits such as Dynet or Chainer, is that its computation graphs are *dynamic*. This will be a key factor simplifying design once we start dealing with more complex models.

2.3 Computation Graphs

2.3.1 Example: The computation graph of a log-linear model

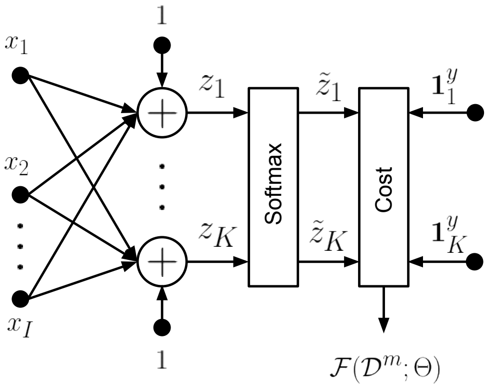


Figure 2.1: Representation of a log-linear model as a computation graph: a composition of linear and non-linear transformations. The classification cost for the m-th training example $\mathcal{D}^m = \{\mathbf{x}, y\}$ is also shown. Note $\mathbf{1}^y$ is an indicator vector of size K with a one in position y and zeros elsewhere.

A computation graph is just a way of expressing compositions of functions with a directed acyclic graph. Fig. 2.1 depicts a log-linear model. Each circle or box corresponds to a node generating one or more outputs by applying some operation over one or more inputs of the preceding nodes. Circles here denote linear transformations, that is weighted sums of the node input plus a bias. The k_{th} node output can be thus described as

$$z_k = \sum_{i=1}^{I} W_{ki} x_i + b_k, \tag{2.1}$$

where W_{ki} and b_k are weights and bias respectively. Squared boxes represent non-linear transformations. Applying a *softmax* function is a way of transforming a K dimensions real-valued vector into a vector of the same dimension where the sum of all components is one. This allows us to consider the output of this node as a probability distribution. The softmax in Fig. 2.1 can be expressed as

$$p(y=k|x) \equiv \tilde{z}_k = \frac{\exp(z_k)}{\sum_{k'=1}^{K} \exp(z_{k'})}. \tag{2.2}$$

Note that in the following sections we will also use $\mathbf{z}$ and $\tilde{\mathbf{z}}$ to denote the output of linear and non-linear functions respectively. By composing Eq. 2.1 and Eq. 2.2 we obtain a log-linear model similar to the one we saw on Chapter 1[4]. This is given by

$$p(y=k|x) = \frac{1}{Z(\mathbf{W}, \mathbf{b}, \mathbf{x})} \exp\left(\sum_{i=1}^{I} W_{ki} x_i + b_k\right), \tag{2.3}$$

[4] There are some differences with respect to Eq.1.26, like the use of a bias $\mathbf{b}$. Also, if we consider the binary joint feature mapping $f(x,y) = g(x) \otimes e_y$ of Eq.1.16, the maximum entropy classifier in Eq.1.26 becomes a special case of Eq.2.3, in which the feature vector $\mathbf{x}$ only takes binary values and the bias parameters in $\mathbf{b}$ are all set to zero.

where

$$Z(\mathbf{W}, \mathbf{b}, \mathbf{x}) = \sum_{k'=1}^{K} \exp\left(\sum_{i=1}^{I} W_{k'i} x_i + b_{k'}\right) \tag{2.4}$$

is the partition function ensuring that all output values sum to one. The model thus receives a feature vector $\mathbf{x} \in \mathbb{R}^I$ and assigns a probability over $y \in 1 \cdots K$ possible class indices. It is parametrized by weights and bias $\Theta = \{\mathbf{W}, \mathbf{b}\}$, with $\mathbf{W} \in \mathbb{R}^{K \times I}$ and $\mathbf{b} \in \mathbb{R}^K$.

2.3.2 Stochastic Gradient Descent: a refresher

As we saw on day one, the parameters of a log linear model $\Theta = \{\mathbf{W}, \mathbf{b}\}$ can be learned with Stochastic Gradient Descent (SGD). To apply SGD we first need to define an error function that measures how good we are doing for any given parameter values. To remain close to the maximum entropy example, we will use as cost function the average minus posterior probability of the correct class, also known as the Cross-Entropy (CE) criterion. Bear in mind, however, that we could pick other non-linear functions and cost functions that do not have a probabilistic interpretation. For example, the same principle could be applied to a regression problem where the cost is the Mean Square Error (MSE). For a training data-set $\mathcal{D} = \{(\mathbf{x}^1, y^1), \ldots, (\mathbf{x}^M, y^M)\}$ of M examples, the CE cost function is given by

$$\mathcal{F}(\mathcal{D}; \Theta) = -\frac{1}{M} \sum_{m=1}^{M} \log p(y^m = k(m) | \mathbf{x}^m), \tag{2.5}$$

where $k(m)$ is the correct class index for the m-th example. To learn the parameters of this model with SGD, all we need to do is compute the gradient of the cost $\nabla\mathcal{F}$ with respect to the parameters of the model and iteratively update our parameter estimates as

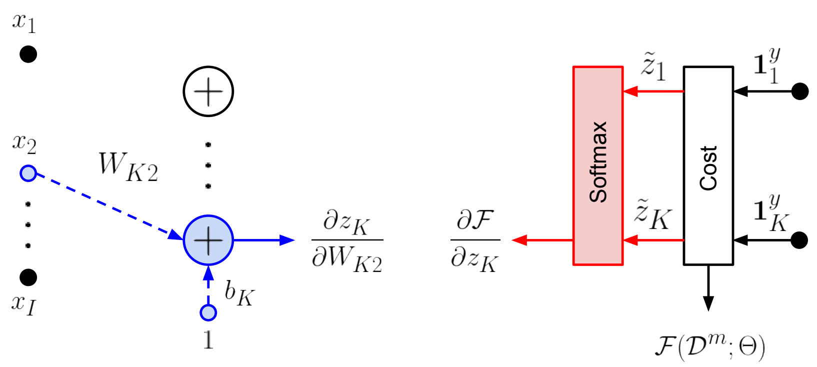
$$\mathbf{W} \leftarrow \mathbf{W} - \eta \nabla_{\mathbf{W}} \mathcal{F} \tag{2.6}$$

and

$$\mathbf{b} \leftarrow \mathbf{b} - \eta \nabla_{\mathbf{b}} \mathcal{F}, \tag{2.7}$$

where η is the learning rate. Note that in practice we will use a mini-batch of examples as opposed to the whole train set. Very often, more elaborated learning rules as e.g. momentum or Adagrad are used. Bear in mind that, in general, these still require the computation of the gradients as the main step. The reasoning here outlined will also be applicable to those.

2.3.3 Deriving Gradients in Computation Graphs using the Chain Rule



x_1
x_2
x_I
W_{K2}
b_K
1
$\frac{\partial z_K}{\partial W_{K2}}$
$\frac{\partial \mathcal{F}}{\partial z_K}$
Softmax
Cost
$\tilde{z}_1$
$\tilde{z}_K$
$\mathbf{1}_1^y$
$\mathbf{1}_K^y$
$\mathcal{F}(\mathcal{D}^m; \Theta)$

Figure 2.2: Forward-pass (blue) and Backpropagation (red) calculations to estimate the gradient of weight W_{K2} and bias b_K of a log-linear model.

The expressions for $\nabla\mathcal{F}$ are well known in the case of log-linear models. However, for the sake of the introduction to deep learning, we will show how they can be derived by exploiting the decomposition of the cost function into the computational graph seen in the last section (and represented in Fig. 2.1). To simplify notation, and without loss of generality, we will work with the classification cost of an individual example

$$\mathcal{F}(\mathcal{D}^m;\Theta) = -\log p(y^m = k(m)|\mathbf{x}^m), \tag{2.8}$$

where $\mathcal{D}^m = \{(\mathbf{x}^m, y^m)\}$.

Lets start by computing the element (k,i) of the gradient matrix $\nabla_{\mathbf{W}}\mathcal{F}(\mathcal{D};\Theta)$, which contains the partial derivative with respect to the weight W_{ki}. To do this, we invoke the **chain rule** to split the derivative calculation into two terms at variable $z_{k'}$ (Eq.2.1) with $k' = 1 \cdots K$

$$\frac{\partial \mathcal{F}(\mathcal{D}^m;\Theta)}{\partial W_{ki}} = \sum_{k'=1}^{K} {\color{red}\frac{\partial \mathcal{F}(\mathcal{D}^m;\Theta)}{\partial z_{k'}}} {\color{blue}\frac{\partial z_{k'}}{\partial W_{ki}}}. \tag{2.9}$$

We have thus transformed the problem of computing the derivative into computing two easier derivatives. We start by the right-most term. The relation between $z_{k'}$ and W_{ki} is given in Eq. 2.1. Since $z_{k'}$ only depends on the weight W_{ki} in a linear way, the second derivative in Eq.2.12 is given by

$${\color{blue}\frac{\partial z_{k'}}{\partial W_{ki}}} = \frac{\partial}{\partial W_{ki}}\left(\sum_{i'=1}^{I} W_{k'i'} x_{i'}^m + b_{k'}\right) = \begin{cases} x_i^m & \text{if } k = k' \\ 0 & \text{otherwise .} \end{cases} \tag{2.10}$$

For the left-most term, the relation between $\mathcal{F}(\mathcal{D}^m;\Theta)$ and z_k is given by Eq. 2.2 together with 2.8

$${\color{red}\frac{\partial \mathcal{F}(\mathcal{D}^m;\Theta)}{\partial z_k}} = -\frac{\partial}{\partial z_k}\left(z_{k(m)} - \log\left(\sum_{k'=1}^{K} \exp(z_{k'})\right)\right) = \begin{cases} -(1-\tilde{z}_k) & \text{if } k = k(m) \\ -(-\tilde{z}_k) & \text{otherwise .} \end{cases} \tag{2.11}$$

Bringing the two parts together, we obtain

$$\frac{\partial \mathcal{F}(\mathcal{D}^m;\Theta)}{\partial W_{ki}} = \begin{cases} -(1-\tilde{z}_k)x_i^m & \text{if } k = k(m) \\ -(-\tilde{z}_k)x_i^m & \text{otherwise .} \end{cases}. \tag{2.12}$$

From the formula for each element and a single example, we can now obtain the gradient matrix for a batch of M examples by simply averaging and expressing the previous equations in vector form as follows

$$\nabla_{\mathbf{W}}\mathcal{F}(\mathcal{D};\Theta) = -\frac{1}{M}\sum_{m=1}^{M}\left(\mathbf{1}^{y^m} - \tilde{\mathbf{z}}^m\right)(\mathbf{x}^m)^T. \tag{2.13}$$

Here $\mathbf{1}^{y^m} \in \mathbb{R}^K$ is a vector of zeros with a one in $y^m = k(m)$, which is the index of the correct class for the example m.

In order to compute the derivatives of the cost function with respect to the bias parameters b_k, we only need to compute one additional derivative

$$\frac{\partial z_{k'}}{\partial b_k} = \begin{cases} 1 & \text{if } k = k' \\ 0 & \text{otherwise .} \end{cases} \tag{2.14}$$

This leads us to the last gradient expression

$$\nabla_{\mathbf{b}}\mathcal{F}(\mathcal{D};\Theta) = -\frac{1}{M}\sum_{m=1}^{M}\left(\mathbf{1}^{y^m} - \tilde{\mathbf{z}}^m\right). \tag{2.15}$$

An important consequence of the previous derivation is the fact that each gradient of the parameters $\nabla_{\mathbf{W}}\mathcal{F}(\mathcal{D};\Theta)$ and $\nabla_{\mathbf{b}}\mathcal{F}(\mathcal{D};\Theta)$ can be computed from two terms, displayed with corresponding colours in Fig. 2.2:

1. The derivative of the cost with respect to the k_{th} output of the linear transformation $\partial\mathcal{F}(\mathcal{D}^m;\Theta)/\partial z_k$, denoted in red. This is, in effect, the error that we propagate *backwards* from the cost layer.

2. The derivative of the forward-pass up to the linear transformation applying the weight $\partial z_k/\partial W_{ki}$, denoted in blue. This is always equal to the input multiplying that weight or one in the case of the bias.

This is the key to the Backpropagation algorithm as we will see in the next Section 2.4.

Exercise 2.1 *To ease-up the upcoming implementation exercise, examine and comment the following implementation of a log-linear model and its gradient update rule. Start by loading Amazon sentiment corpus used in day 1*

```
import lxmls.readers.sentiment_reader as srs
from lxmls.deep_learning.utils import AmazonData
corpus=srs.SentimentCorpus("books")
data = AmazonData(corpus=corpus)
```

Compare the following numpy implementation of a log-linear model with the derivations seen in the previous sections. Introduce comments on the blocks marked with # relating them to the corresponding algorithm steps.

```
from lxmls.deep_learning.utils import Model, glorot_weight_init, index2onehot, logsumexp
import numpy as np

class NumpyLogLinear(Model):

    def __init__(self, **config):

        # Initialize parameters
        weight_shape = (config['input_size'], config['num_classes'])
        # after Xavier Glorot et al
        self.weight = glorot_weight_init(weight_shape, 'softmax')
        self.bias = np.zeros((1, config['num_classes']))
        self.learning_rate = config['learning_rate']

    def log_forward(self, input=None):
        """Forward pass of the computation graph"""

        # Linear transformation
        z = np.dot(input, self.weight.T) + self.bias

        # Softmax implemented in log domain
        log_tilde_z = z - logsumexp(z, axis=1, keepdims=True)

        return log_tilde_z

    def predict(self, input=None):
        """Prediction: most probable class index"""
        return np.argmax(np.exp(self.log_forward(input)), axis=1)

    def update(self, input=None, output=None):
        """Stochastic Gradient Descent update"""

        # Probabilities of each class
        class_probabilities = np.exp(self.log_forward(input))
        batch_size, num_classes = class_probabilities.shape

        # Error derivative at softmax layer
        I = index2onehot(output, num_classes)
        error = (class_probabilities - I) / batch_size

        # Weight gradient
        gradient_weight = np.zeros(self.weight.shape)
        for l in range(batch_size):
            gradient_weight += np.outer(error[l, :], input[l, :])

        # Bias gradient
        gradient_bias = np.sum(error, axis=0, keepdims=True)

        # SGD update
        self.weight = self.weight - self.learning_rate * gradient_weight
        self.bias = self.bias - self.learning_rate * gradient_bias
```

Instantiate model and data classes. Check the initial accuracy of the model. This should be close to 50% since we are on a binary prediction task and the model is not trained yet.

```python
# Instantiate model
model = NumpyLogLinear(
    input_size=corpus.nr_features,
    num_classes=2,
    learning_rate=0.05
)

# Define number of epochs and batch size
num_epochs = 10
batch_size = 30

# Instantiate data iterators
train_batches = data.batches('train', batch_size=batch_size)
test_set = data.batches('test', batch_size=None)[0]

# Check initial accuracy
hat_y = model.predict(input=test_set['input'])
accuracy = 100*np.mean(hat_y == test_set['output'])
print("Initial accuracy %2.2f %%" % accuracy)
```

Train the model with simple batch stochastic gradient descent. Be sure to understand each of the steps involved, including the code running inside of the model class. We will be wokring on a more complex version of the model in the upcoming exercise.

```python
# Epoch loop
for epoch in range(num_epochs):

    # Batch loop
    for batch in train_batches:
        model.update(input=batch['input'], output=batch['output'])

    # Prediction for this epoch
    hat_y = model.predict(input=test_set['input'])

    # Evaluation
    accuracy = 100*np.mean(hat_y == test_set['output'])
    print("Epoch %d: accuracy %2.2f %%" % (epoch+1, accuracy))
```

2.4 Going Deeper than Log-linear by using Composition

2.4.1 The Multilayer Perceptron or Feed-Forward Network

We have seen that just by using the chain rule we can easily compute gradients for compositions of two functions (one non-linear and one linear). However, there was nothing in the derivation that would stop us from composing more than two functions. The algorithm in 7 describes the Multi-Layer Perceptron (MLP) or Feed-Forward (FF) network. In a similar fashion to the log-linear model, the MLP/FF can be expressed as a computation graph and is displayed in Fig. 2.3. Take into account the following:

- MLPs/FFs are characterized by applying functions in a set of layers subsequently to a single input. This characteristic is also shared by convolutional networks, although the latter also have parameter tying constraints.

- The non-linearities in the intermediate layers are usually one-to-one transformations. The most typical are the sigmoid, hyperbolic tangent and the rectified linear unit (ReLU).

- The output non-linearity is determined by the output to be estimated. In order to estimate probability distributions the softmax is typically used. For regression problems a last linear layer is used instead.

Algorithm 7 Forward pass of a Multi-Layer Perceptron (MLP) or Feed-Forward (FF) network

```
input: Initial parameters for an MLP of $N$ layers $\Theta = \{\mathbf{W}^1, \mathbf{b}^1, \cdots \mathbf{W}^N, \mathbf{b}^N\}$
input: Input data vector $\tilde{\mathbf{z}}^0 \equiv \mathbf{x}$.
for $n = 1$ to $N-1$ do
    Apply linear transformation
        $$z_j^n = \sum_{i=1}^{I} W_{ji}^n \tilde{z}_i^{n-1} + b_j^n$$
    Apply non-linear transformation e.g. sigmoid (hereby denoted $\sigma()$)
        $$\tilde{z}_j^n = \sigma(z_j^n) = \frac{1}{1+\exp(-z_j^n)}$$
end for
Apply final linear transformation
    $$z_k^N = \sum_{j=1}^{J} W_{kj}^N \tilde{z}_j^{N-1} + b_k^N$$
Apply final non-linear transformation e.g. softmax
    $$p(y=k|x) \equiv \tilde{z}_k^N = \frac{\exp(z_k^N)}{\sum_{k'=1}^{K} \exp(z_{k'}^N)}$$
```

2.4.2 Backpropagation: an overview

For the examples in this chapter we will consider the case in which we are estimating a distribution over classes, thus we will use the CE cost function (Eq. 2.5).

To compute the gradient with respect the parameters of the n-th layer, we just need to apply the chain rule as in the previous section, consecutively. Fortunately, we do not need to repeat this procedure for each layer as it is easy to spot a recursive rule (the Backpropagation recursion) that is valid for many neural models, including feed-forward networks (such as MLPs) as well as recurrent neural networks (RNNs) with minor modifications. The Backpropagation method, which is given in Algorithm 8 for the case of an MLP, consists of the following steps:

- The forward pass step, where the input signal is injected though the network in a forward fashion (see Alg. 7)
- The Backpropagation step, where the derivative of the cost function (also called error) is injected back through the network and backpropagated according to the derivative rules (see steps 8-17 in Alg. 8)

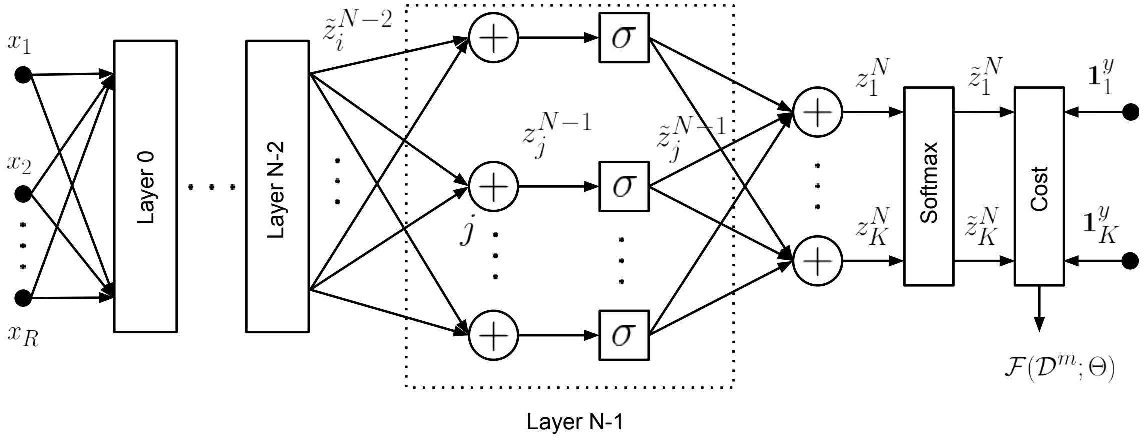


Figure 2.3: Representation of a Multi-Layer Perceptron (MLP) or Feed-Forward (FF) network as a computation graph. The classification cost for the m-th training example $\mathcal{D}^m = \{\mathbf{x}, y\}$ is also shown.

- Finally, the gradients with respect to the parameters are computed by multiplying the input signal from the forward pass and the backpropagated error signal, at the corresponding places in the network (step 18 in Alg. 8)
- Given the gradients computed in the previous step, the model weights can then be easily update according to a specified learning rule (step 19 in Alg. 8 uses a mini-batch SGD update rule).

The main step of the method is the Backpropagation step, where one has to compute the Backpropagation recursion rules for a specific network. The next section presents a careful deduction of these recursion rules, for the present MLP model.

2.4.3 Backpropagation: deriving the rule for a feed forward network

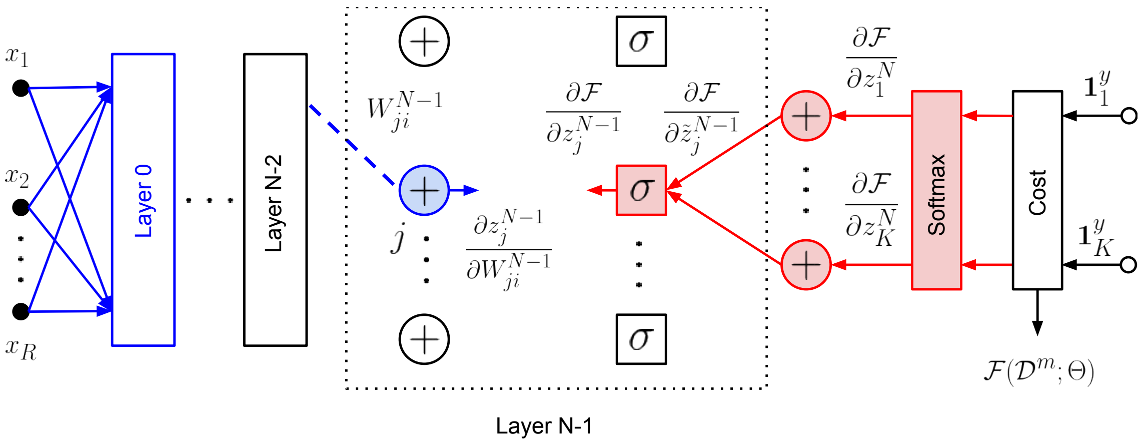

x_1
x_2
x_R
Layer 0
Layer N-2
W_{ji}^{N-1}
j
$\frac{\partial \mathcal{F}}{\partial z_j^{N-1}}$
$\frac{\partial \mathcal{F}}{\partial \tilde{z}_j^{N-1}}$
$\frac{\partial z_j^{N-1}}{\partial W_{ji}^{N-1}}$
σ
$\frac{\partial \mathcal{F}}{\partial z_1^N}$
$\frac{\partial \mathcal{F}}{\partial z_K^N}$
Softmax
Cost
$\mathbf{1}_1^y$
$\mathbf{1}_K^y$
$\mathcal{F}(\mathcal{D}^m;\Theta)$
Layer N-1

Figure 2.4: Forward-pass (blue) and Backpropagation (red) calculations to estimate the gradient of weight W_{ji} at layer $N-1$ of a MLP.

In a generic MLP we would like to compute the values of all parameters $\Theta = \{\mathbf{W}^1, \mathbf{b}^1, \cdots \mathbf{W}^N, \mathbf{b}^N\}$. As explained previously, we will thus need to compute the backpropagated error at each node $\partial \mathcal{F}(\mathcal{D}^m;\Theta)/\partial z_k^n$, and the corresponding derivative for the forward-pass $\partial z_k^n/\partial W_{ki}$, for $n = 1 \cdots N$. Fortunately, it is easy to spot a recursion that will allow us to compute these values for each node, given all its child nodes. To spot it, we can start trying to compute the gradients one layer before the output layer (see Fig. 2.4), i.e. layer $N-1$. We start by splitting at with respect to the output of the linear layer at $N-1$

$$\frac{\partial \mathcal{F}(\mathcal{D}^m;\Theta)}{\partial W_{ji}^{N-1}} = \sum_{j'=1}^{J} \frac{\partial \mathcal{F}(\mathcal{D}^m;\Theta)}{\partial z_{j'}^{N-1}} \frac{\partial z_{j'}^{N-1}}{\partial W_{ji}^{N-1}} = \frac{\partial \mathcal{F}(\mathcal{D}^m;\Theta)}{\partial z_{j}^{N-1}} \frac{\partial z_{j}^{N-1}}{\partial W_{ji}^{N-1}} \tag{2.16}$$

where, as in the case of the log-linear model, we have used the fact that the ouput of the linear layer z_j^{N-1} only depends on W_{ji}^{N-1}. We now pick the left-most factor and apply the chain rule to split by the output of the non-linear layer $\tilde{z}_j^{N-1}$. Assuming that the non linear transformation is one-to-one, as e.g. a sigmoid, tanh, relu we have

$$\frac{\partial \mathcal{F}(\mathcal{D}^m;\Theta)}{\partial z_{j}^{N-1}} = \left(\sum_{j'=1}^{J} \frac{\partial \mathcal{F}(\mathcal{D}^m;\Theta)}{\partial \tilde{z}_{j'}^{N-1}} \frac{\partial \tilde{z}_{j'}^{N-1}}{\partial z_{j}^{N-1}} \right) = \frac{\partial \mathcal{F}(\mathcal{D}^m;\Theta)}{\partial \tilde{z}_{j}^{N-1}} \frac{\partial \tilde{z}_{j}^{N-1}}{\partial z_{j}^{N-1}} \tag{2.17}$$

To spot a recursion we only need to apply the chain rule a third time. The next variable to split by is the linear output of layer N, z_j^N. By looking at Fig. 2.4, it is clear that the derivatives at each node in layer $N-1$ will depend on all values of layer N. For the linear layer the summation won't go away yielding

$$\frac{\partial \mathcal{F}(\mathcal{D}^m;\Theta)}{\partial z_{j}^{N-1}} = \left(\sum_{k'=1}^{K} \frac{\partial \mathcal{F}(\mathcal{D}^m;\Theta)}{\partial z_{k'}^{N}} \frac{\partial z_{k'}^{N}}{\partial \tilde{z}_{j}^{N-1}} \right) \frac{\partial \tilde{z}_{j}^{N-1}}{\partial z_{j}^{N-1}} \tag{2.18}$$

If we call the derivative of the error with respect to the N_{th} linear layer output as

$$e_k^N = \frac{\partial \mathcal{F}(\mathcal{D}^m;\Theta)}{\partial z_k^N} \tag{2.19}$$

it is easy to deduce from Eqs. 2.17, 2.18 that

$$e_j^{N-1} = \left(\sum_{k'=1}^{K} e_{k'}^N \frac{\partial z_{k'}^N}{\partial \tilde{z}_j^{N-1}} \right) \frac{\partial \tilde{z}_j^{N-1}}{\partial z_j^{N-1}}. \tag{2.20}$$

coming back to the original 2.16 we obtain the formula for the update of each of the weights and bias

$$\frac{\partial \mathcal{F}(\mathcal{D}^m;\Theta)}{\partial W_{ji}^{N-1}} = e_j^{N-1} \tilde{z}_i^{N-2}, \quad \frac{\partial \mathcal{F}(\mathcal{D}^m;\Theta)}{\partial b_j^{N-1}} = e_j^{N-1} \tag{2.21}$$

These formulas are valid for any FF network with hidden layers using one-to-one non-linearities. For the network described in Algorithm 7 we have

$$\mathbf{e}^N = \mathbf{1}^y - \tilde{\mathbf{z}}^N, \quad \frac{\partial z_{k'}^N}{\partial \tilde{z}_j^{N-1}} = W_{k'j}^N \quad \text{and} \quad \frac{\partial \tilde{z}_j^n}{\partial z_j^n} = \tilde{z}_j^n \cdot (1 - \tilde{z}_j^n) \quad \text{with} \quad n \in \{1 \cdots N-2\} \tag{2.22}$$

A more detailed version can be seen in Algorithm 8

2.4.4 Backpropagation as a general rule

Once we get comfortable with the derivation of Backpropagation for the FF, it is simple to see that expanding to generic computations graphs is trivial. If we wanted to change the sigmoid non-linearity by a Rectified Linear Unit (ReLU) we would only need to change forward and Backpropagation derivative of the hidden non-linearities as

$$\tilde{z}_j = \begin{cases} z_j & \text{if } z_j >= 0 \\ 0 & \text{otherwise .} \end{cases} \qquad \frac{\partial \tilde{z}_j}{\partial z_j} = \begin{cases} 1 & \text{if } z_j > 0 \\ 0 & \text{otherwise .} \end{cases} \tag{2.23}$$

More importantly, Backpropagation can be always defined as a direct acyclic graph with the *reverse* direction of the forward-pass, where at each node we apply the transpose of the Jacobian of each linear or non linear transformation. Coming back to Eq. 2.22 we have the vector formula

$$\mathbf{e}^{N-1} = \left(\mathrm{J}_{\tilde{z}}^{N-1}\right)^T \left(\mathrm{J}_W^N\right)^T \mathbf{e}^N. \tag{2.24}$$

In other words, regardless of the topology of the network and as long as we can compute the forward-pass and the Jacobian of each individual node, we will be able to compute Backpropagation.

Algorithm 8 Mini-batch SGD with Back-Propagation

```
input: Data $\mathcal{D} = \{\mathcal{D}_1, \mathcal{D}_2, ..., \mathcal{D}_B\}$ split into $B$ mini-batches of size $M'$, MLP of $N$ layers, with parameters $\Theta = \{\mathbf{W}^1, \mathbf{b}^1, \cdots \mathbf{W}^N, \mathbf{b}^N\}$, number of rounds $T$, learning rate $\eta$
initialize parameters $\Theta$ randomly
for $t = 1$ to $T$ do
  for $b = 1$ to $B$ do
    for $m = 1$ to $M'$ do
      Compute the forward pass for each of the $M'$ examples in batch $b$; keep not only $p(y^m|\mathbf{x}^m) \equiv \tilde{\mathbf{z}}^{m,N}$ but also all the intermediate non-linear outputs $\tilde{\mathbf{z}}^{m,1} \cdots \tilde{\mathbf{z}}^{m,N}$.
    end for
    for $n = N$ to $1$ do
      if n==N then
        for $m = 1$ to $M'$ do
          Initialize the error at last layer, for each example $m$. For the softmax with CE cost this is given by:
            $$\mathbf{e}^{m,N} = \left(\mathbf{1}_{k(m)} - \tilde{\mathbf{z}}^{m,N}\right).$$
        end for
      else
        for $m = 1$ to $M'$ do
          Backpropagate the error through the linear layer, for each example $m$:
            $$\mathbf{e}^{m} = \left((\mathbf{W}^{n+1})^T \mathbf{e}^{m,n+1}\right)$$
          Backpropagate the error through the non-linearity, for the sigmoid this is:
            $$\mathbf{e}^{m,n} = \mathbf{e}^{m} \odot \tilde{\mathbf{z}}^{m,n} \odot (1 - \tilde{\mathbf{z}}^{m,n}),$$
          where $\odot$ is the element-wise product and the 1 is replicated to match the size of $\tilde{\mathbf{z}}^n$.
        end for
      end if
      Compute the gradients using the backpropagated errors and the inputs from the forward pass
        $$\nabla_{\mathbf{W}^n} \mathcal{F}(\mathcal{D}; \Theta) = -\frac{1}{M'} \sum_{m=1}^{M'} \mathbf{e}^{m,n} \cdot \left(\tilde{\mathbf{z}}^{m,n-1}\right)^T,$$
        $$\nabla_{\mathbf{b}^n} \mathcal{F}(\mathcal{D}; \Theta) = -\frac{1}{M'} \sum_{m=1}^{M'} \mathbf{e}^{m,n}.$$
      Update the parameters
        $$\mathbf{W}^n \leftarrow \mathbf{W}^n - \eta \nabla_{\mathbf{W^n}} \mathcal{F},$$
        $$\mathbf{b}^n \leftarrow \mathbf{b}^n - \eta \nabla_{\mathbf{b^n}} \mathcal{F}.$$
    end for
  end for
end for
```

Exercise 2.2 *Instantiate the feed-forward model class and optimization parameters. This models follows the architecture described in Algorithm 7.*

```
# Model
geometry = [corpus.nr_features, 20, 2]
activation_functions = ['sigmoid', 'softmax']

# Optimization
learning_rate = 0.05
num_epochs = 10
batch_size = 30

# Instantiate model
from lxmls.deep_learning.numpy_models.mlp import NumpyMLP
model = NumpyMLP(
    geometry=geometry,
    activation_functions=activation_functions,
    learning_rate=learning_rate
)
```

Open the code for this model. This is located in 'lxmls/deep_learning/numpy_models/mlp.py'. Implement the method 'backpropagation()' in the class 'NumpyMLP' using Backpropagation recursion that we just saw.

As a first step focus on getting the gradients of each layer, one at a time. Use the code below to plot the loss values for the study weight and perturbed versions.

```
from lxmls.deep_learning.mlp import get_mlp_parameter_handlers, get_mlp_loss_range

# Get functions to get and set values of a particular weight of the model
get_parameter, set_parameter = get_mlp_parameter_handlers(
    layer_index=1,
    is_bias=False,
    row=0,
    column=0
)

# Get batch of data
batch = data.batches('train', batch_size=batch_size)[0]

# Get loss and weight value
current_loss = model.cross_entropy_loss(batch['input'], batch['output'])
current_weight = get_parameter(model.parameters)

# Get range of values of the weight and loss around current parameters values
weight_range, loss_range = get_mlp_loss_range(model, get_parameter, set_parameter, batch)
```

Once you have implemented at least the gradient of the last layer. You can start checking if the values match

```
gradients = model.backpropagation(batch['input'], batch['output'])
current_gradient = get_parameter(gradients)
```

Now you can plot the values of the loss around a given parameters value versus the gradient. If you have implemented this correctly the gradient should be tangent to the loss at the current weight value, see Figure 2.5. Once you have completed the exercise, you should be able to plot also the gradients of the other layers. Take into account that the gradients for the first layer will only be non zero for the indices of words present in the batch. You can locate this using.

```
batch['input'][0].nonzero()
```

Copy the following code for plotting

```python
%matplotlib inline  # for jupyter notebooks
import matplotlib.pyplot as plt
# Plot empirical
plt.plot(weight_range, loss_range)
plt.plot(current_weight, current_loss, 'xr')
plt.ylabel('loss value')
plt.xlabel('weight value')
# Plot real
h = plt.plot(
    weight_range,
    current_gradient*(weight_range - current_weight) + current_loss,
    'r--'
)
```

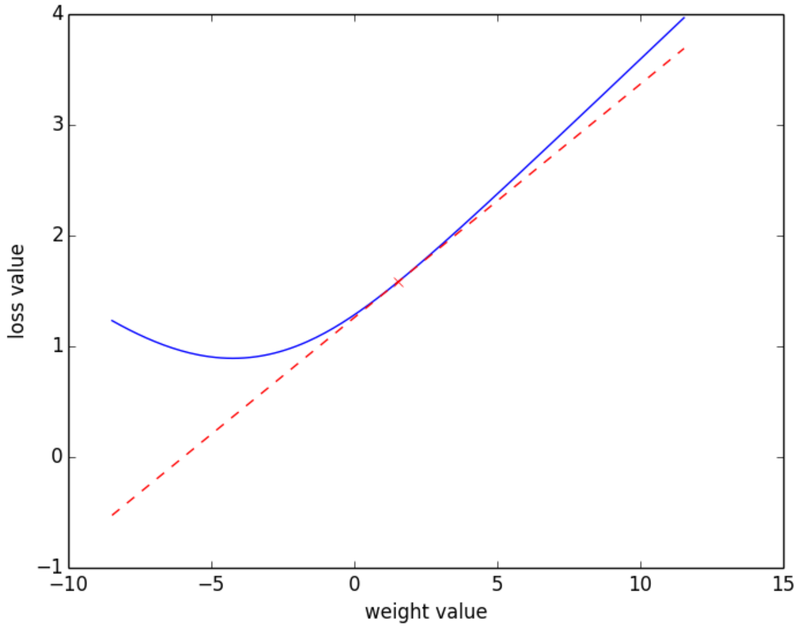


Figure 2.5: Values of the loss (blue) and gradient (dashed red) for a given weight of the network, as well loss values for small perturbations of the weight.

After you have ensured that your Backpropagation algorithm is correct, you can train a model with the data we have.

```python
# Get batch iterators for train and test
train_batches = data.batches('train', batch_size=batch_size)
test_set = data.batches('test', batch_size=None)[0]

# Epoch loop
for epoch in range(num_epochs):

    # Batch loop
    for batch in train_batches:
        model.update(input=batch['input'], output=batch['output'])

    # Prediction for this epoch
    hat_y = model.predict(input=test_set['input'])

    # Evaluation
    accuracy = 100*np.mean(hat_y == test_set['output'])
    print("Epoch %d: accuracy %2.2f %%" % (epoch+1, accuracy))
```

2.4.5 Some final reflections on Backpropagation

If you are new to the neural network topic, this is about the most important piece of theory you should learn about deep learning. Here are some reflections that you should keep in mind.

- Backpropagation allows us in principle to compute the gradients for any differentiable computation graph.
- We only need to know the forward-pass and Jacobian of each individual node in the network to implement Backpropagation.
- Learning guarantees are however weaker than for expectation maximization or convex optimization algorithms.
- In practice optimization will often get trapped on local minima and exhibit high variance in performance for small changes.

2.5 Deriving gradients and GPU code with Pytorch

2.5.1 An Introduction to Pytorch and Computation Graph Toolkits

As you may have observed, the speed of SGD training for MLPs slows down considerably when we increase the number of layers. One reason for this is that the code that we use here is not very optimized. It is thought for you to learn the basic principles. Even if the code was more optimized, it would still be very slow for reasonable network sizes. The cost of computing each linear layer is proportional to the dimensionality of the previous and current layers, which in most cases will be rather large.

For this reason most deep learning applications use Graphics Processing Units (GPU) in their computations. This specialized hardware is normally used to accelerate computer graphics, but can also be used for some computation intensive tasks like matrix multiplications. However, we need to deal with specific interfaces and operations in order to use a GPU. This is where Pytorch comes in. Pytorch is a computation graph toolkit with following nice features

- Automatic differentiation. We only need to express the computation graph of the forward pass. Pytorch will compute the gradients for us.
- GPU integration: The code will be ready to work on a GPU.
- An active community focused on the application of Pytorch to Deep Learning.
- Dynamic computation graphs. This allows us to change the computation graph within each update.

Note that all of these properties are separately available in other toolkits. Dynet has very good dynamic graph functionality and cpu performance, Tensor Flow is backed by Google and has a large community, Amazon's MXNet of Microsoft's CNTK are also competing to play a central role. It is hard to say at this point which toolkit will be the best option in the future. At this point we chose Pytorch because it strikes a balance between a strong community, ease of use and dynamic computation graphs. Also take into account that transiting from a toolkit to another is not very complicated, as the primitives are relatively similar across them.

In general, and compared to numpy, computation graph toolkits are less easy to use. In the case of Pytorch we will have to consider following aspects

- Pytorch types are less flexible than numpy arrays since they have to act on data stored on the GPU. Casting of all variables to Pytorch types will be often a source of errors.
- Not all operations available in numpy are available on Pytorch. Also the semantics of the function may differ.
- Despite being a big improvement compared to the early toolkits like Theano, Pytorch errors can still be difficult to track sometimes.
- As we will see, Pytorch has a good GPU performance, but its CPU performance is not great. Particularly at small sizes.

Exercise 2.3 *In order to learn the differences between a numpy and a Pytorch implementation, explore the reimplementation of Ex. 2.1 in Pytorch. Compare the content of each of the functions, in particular the forward() and update methods(). The comments indicated as IMPORTANT will highlight common sources of errors.*

```
import torch


class PytorchLogLinear(Model):

  def __init__(self, **config):

    # Initialize parameters
    weight_shape = (config['input_size'], config['num_classes'])
    # after Xavier Glorot et al
    weight_np = glorot_weight_init(weight_shape, 'softmax')
    self.learning_rate = config['learning_rate']

    # IMPORTANT: Cast to pytorch format
    self.weight = torch.from_numpy(weight_np).float()
    self.weight.requires_grad = True

    self.bias = torch.zeros(1, config['num_classes'], requires_grad=True)

    self.log_softmax = torch.nn.LogSoftmax(dim=1)
    self.loss_function = torch.nn.NLLLoss()

  def _log_forward(self, input=None):
    """Forward pass of the computation graph in logarithm domain (pytorch)"""

    # IMPORTANT: Cast to pytorch format
    input = torch.from_numpy(input).float()

    # Linear transformation
    z =  torch.matmul(input, torch.t(self.weight)) + self.bias

    # Softmax implemented in log domain
    log_tilde_z = self.log_softmax(z)

    # NOTE that this is a pytorch class!
    return log_tilde_z

  def predict(self, input=None):
    """Most probable class index"""
    log_forward = self._log_forward(input).data.numpy()
    return np.argmax(log_forward, axis=1)

  def update(self, input=None, output=None):
    """Stochastic Gradient Descent update"""

    # IMPORTANT: Class indices need to be casted to LONG
    true_class = torch.from_numpy(output).long()

    # Compute negative log-likelihood loss
    loss = self.loss_function(self._log_forward(input), true_class)

    # Use autograd to compute the backward pass.
    loss.backward()

    # SGD update
    self.weight.data -= self.learning_rate * self.weight.grad.data
    self.bias.data -= self.learning_rate * self.bias.grad.data

    # Zero gradients
    self.weight.grad.data.zero_()
    self.bias.grad.data.zero_()
    return loss.data.numpy()
```

Once you understand the model you can instantiate it and run it using the standard training loop we have used on previous exercises.

```
# Instantiate model
model = PytorchLogLinear(
    input_size=corpus.nr_features,
    num_classes=2,
    learning_rate=0.05
)
```

Exercise 2.4 *As the final exercise today implement the log_forward() method in lxmls/deep_learning/pytorch_models/mlp.py. Use the previous exercise as reference. After you have completed this you can run both systems for comparison.*

```
# Model
geometry = [corpus.nr_features, 20, 2]
activation_functions = ['sigmoid', 'softmax']

# Instantiate model
import numpy as np
from lxmls.deep_learning.pytorch_models.mlp import PytorchMLP
model = PytorchMLP(
    geometry=geometry,
    activation_functions=activation_functions,
    learning_rate=0.05
)
```

Day 3

Sequence Models

Today's class will be focused on advanced deep learning concepts, mainly Recurrent Neural Networks (RNNs). In the first day we saw how the chain-rule allowed us to compute gradients for arbitrary computation graphs. Today we will see that we can still do this for more complex models like Recurrent Neural Networks (RNNs). In these models we will input data in different points of the graph, which will correspond to different time instants. The key factor to consider is that, for a fixed number of time steps, this is still a computation graph and all what we saw on the first day applies with no need for extra math.

If you managed to finish the previous day completely you should aim at finishing this as well. If you still have pending exercises from the first day e.g. the Pytorch part. It is recommended that you try to solve them first and then continue with this day.

3.1 Recurrent Neural Networks: Backpropagation Through Time

3.1.1 Feed Forward Networks Unfolded in Time

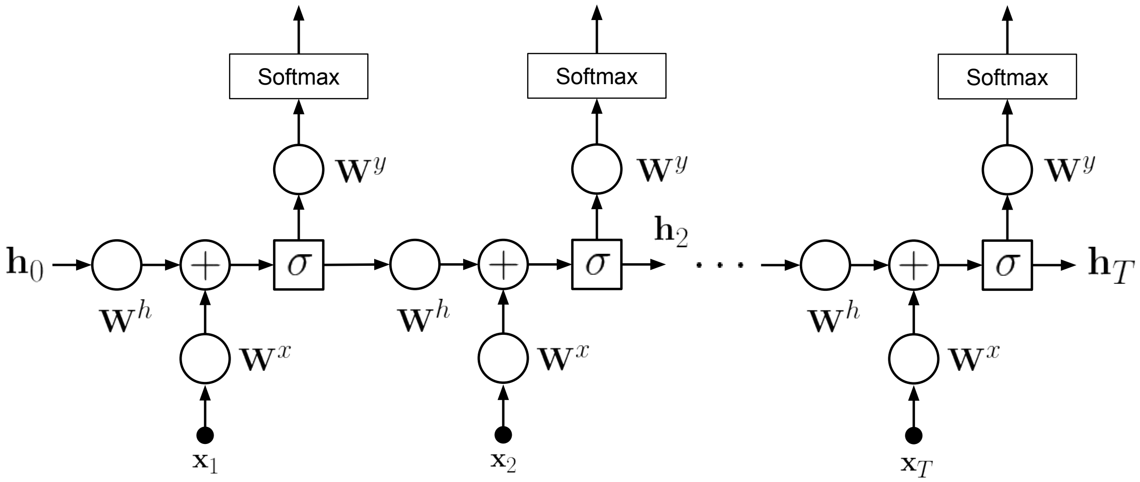


Figure 3.1: The simplest RNN can be seen as replicating a single hidden-layer FF network T times and passing the intermediate hidden variable $\mathbf{h}_t$ across different steps. Note that all nodes operate over vector inputs e.g,. $\mathbf{x}_t \in \mathbb{R}^I$. Circles indicate matrix multiplications.

We have seen already Feed Forward (FF) networks. These networks are ill suited to learn variable length patterns since they only accept inputs of a fixed size. In order to learn sequences using neural networks, we need therefore to define some architecture that is able to process variable length inputs. Recurrent Neural Networks (RNNs) solve this problem by unfolding the computation graph in time. In other words, the network is replicated as many times as it is necessary to cover the sequence to be modeled. In order to model the sequence one or more connections across different time instants are created. This allows the network to have a memory in time and thus capture complex patterns in sequences. In the simplest model, depicted in Fig. 3.1, and detailed in Algorithm 9, a RNN is created by replicating a single hidden-layer FF network T times and passing the intermediate hidden variable across different steps. The strength of the connection is determined by the weight matrix $\mathbf{W}_h$

3.1.2 Backpropagating through Unfolded Networks

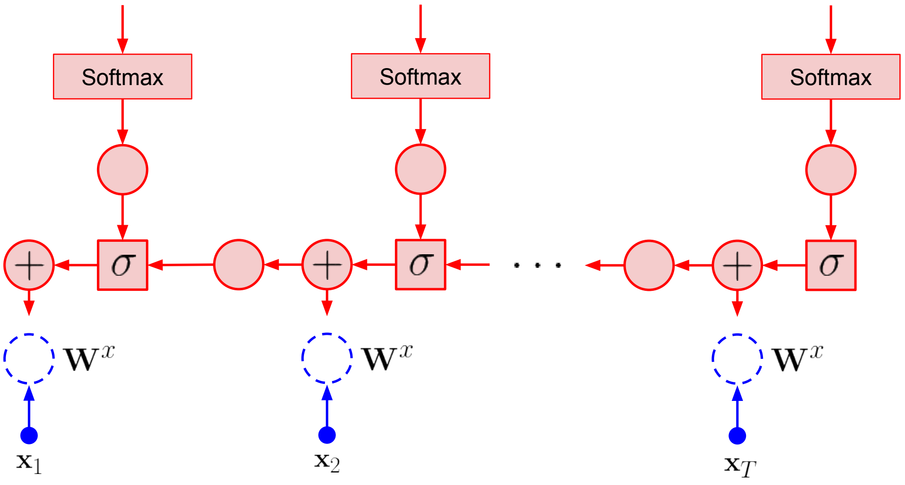


Figure 3.2: Forward-pass (blue) and backpropagated error (red) to the input layer of an RNN. Note that a copy of the error is sent to each output of each sum node $(+)$

It is important to note that there is no formal changes needed to apply backpropagation to RNNs. It concerns applying the chain rule just as it happened with FFs. It is however useful to consider the following properties of derivatives, which are not relevant when dealing with FFs

- When two variables are summed up in the forward-pass, the error is backpropagated to each of the summand sub-graphs
- When unfolding in T steps the same parameters will be copied T times. All updates for each copy are summed up to compute the total gradient.

Despite the lack of formal changes, the fact that we backpropagate an error over the length of the entire sequence often leads to numerical problems. The problem of *vanishing* and *exploding* gradients are a well know limitation. A number of solutions are used to mitigate this issue. One simple, yet inelegant, method is clipping the gradients to a fixed threshold. Another solution is to resort to more complex RNN models that are able to better handle long range dependencies and are less sensitive to this phenomena. It is important to bear in mind, however, that all RNNs still use backpropagation as seen in the previous day, although it is often referred as *Backpropagation through time*.

Exercise 3.1 *Convince yourself that a RNN is just an FF unfolded in time. Complete the backpropagation() method in NumpyRNN class in lxmls/deep_learning/numpy_models/rnn.py and compare it with lxmls/deep_learning/numpy/models/mlp.py.*

To work with RNNs we will use the Part-of-speech data-set.

```
# Load Part-of-Speech data
from lxmls.readers.pos_corpus import PostagCorpusData
data = PostagCorpusData()
```

Algorithm 9 Forward pass of a Recurrent Neural Network (RNN) with embeddings

```
input: Initial parameters for an RNN Input $\Theta = \{\mathbf{W}^e \in \mathbb{R}^{E\times I}, \mathbf{W}^x \in \mathbb{R}^{J\times E}, \mathbf{W}^h \in \mathbb{R}^{J\times J}, \mathbf{W}^y \in \mathbb{R}^{K\times J}\}$ embedding, input, recurrent and output linear transformations respectively.
input: Input data matrix $\mathbf{x} \in \mathbb{R}^{I\times M'}$, representing sentence of $M'$ time steps. Initial recurrent variable $\mathbf{h}_0$.
for $m = 1$ to $M'$ do
    Apply embedding layer
        $$z^e_{dm} = \sum_{i=1}^{I} W^e_{di} x_{im}$$
    Apply linear transformation combining embedding and recurrent signals
        $$z^h_{jm} = \sum_{d=1}^{E} W^x_{jd} z^e_{dm} + \sum_{j'=1}^{J} W^h_{jj'} h_{j'm-1}$$
    Apply non-linear transformation e.g. sigmoid (hereby denoted $\sigma()$)
        $$h_{jm} = \sigma(z^h_{jm}) = \frac{1}{1 + \exp(-z^h_{jm})}$$
end for
Apply final linear transformation to each of the recurrent variables $\mathbf{h}_1 \cdots \mathbf{h}_{M'}$
    $$z^y_{km} = \sum_{j=1}^{J} W^y_{kj} h_{jm}$$
Apply final non-linear transformation e.g. softmax
    $$p(y_m = k|\mathbf{x}_{1:m+1}) = \tilde{z}^y_{km} = \frac{\exp(z^y_{km})}{\sum_{k'=1}^{K} \exp(z^y_{k'm})}$$
```

Algorithm 10 Backpropagation for a Recurrent Neural Network (RNN) with embeddings

```
input: Data $\mathcal{D} = \{\mathcal{D}_1, \mathcal{D}_2, ..., \mathcal{D}_B\}$ split into $B$ sequences (batch size 1) of size $M'$. RNN with parameters $\Theta = \{\mathbf{W}^e \in \mathbb{R}^{E \times I}, \mathbf{W}^x \in \mathbb{R}^{J \times E}, \mathbf{W}^h \in \mathbb{R}^{J \times J}, \mathbf{W}^y \in \mathbb{R}^{K \times J}\}$ embedding, input, recurrent and output linear transformations respectively. Number of rounds $T$, learning rate $\eta$
initialize parameters $\Theta$ randomly
for $t = 1$ to $T$ do
    for $b = 1$ to $B$ do

        for $m = 1$ to $M'$ do
            Compute the forward pass for sentence $b$ ($M'$ time steps); keep not only $\tilde{z}^y_{km}$ but also, the RNN layer activations $h_{jm}$ and the embedding layer activations $z^e_{dm}$.
        end for

        for $m = 1$ to $M'$ do
            Initialize the error at last layer for a CE cost and backpropagate it through the output linear layer:
                $\mathbf{e}^m_y = (\mathbf{W}^y)^T \left(\mathbf{1}_{k(m)} - \tilde{\mathbf{z}}^y_m\right).$
        end for
        Initialize recurrent layer error $\mathbf{e}_r$ to a vector of zeros of size $J$
        for $m = M'$ to 1 do
            Add the recurrent layer backpropagated error and backpropagate through the sigmoid non-linearity:
                $\mathbf{e}^m_h = \left(\mathbf{e}_r + \mathbf{e}^m_y\right) \odot \mathbf{h}_m \odot (1 - \mathbf{h}_m)$
            where $\odot$ is the element-wise product and the 1 is replicated to match the size of $\mathbf{h}^n$.
            Backpropagate the error through the recurrent linear layer
                $\mathbf{e}_r = (\mathbf{W}^h)^T \mathbf{e}^m_h$
            Backpropagate the error through the input linear layer
                $\mathbf{e}^m_e = (\mathbf{W}^x)^T \mathbf{e}^m_h$
        end for

        Compute the gradients using the backpropagated errors and the inputs from the forward pass. For example for the recurrent layer
            $\nabla_{\mathbf{W}^h} \mathcal{F}(\mathcal{D};\Theta) = -\frac{1}{M'} \sum_{m=1}^{M'} \mathbf{e}^m_h \cdot (\mathbf{h}_{m-1})^T ,$
        For the input layer
            $\nabla_{\mathbf{W}^x} \mathcal{F}(\mathcal{D};\Theta) = -\frac{1}{M'} \sum_{m=1}^{M'} \mathbf{e}^m_h \cdot (\mathbf{z}^e_m)^T ,$
        Update the parameters, for example for the input layer
            $\mathbf{W}^x \leftarrow \mathbf{W}^x - \eta \nabla_{\mathbf{W}^x} \mathcal{F},$
    end for
end for
```

Load and configure the NumpyRNN. Remember to use reload if you want to modify the code inside the rnns module

```
# Instantiate RNN
from lxmls.deep_learning.numpy_models.rnn import NumpyRNN
model = NumpyRNN(
    input_size=data.input_size,
    embedding_size=50,
    hidden_size=20,
    output_size=data.output_size,
    learning_rate=0.1
)
```

As in the case of the feed-forward networks you can use the following setup to test step by step the implementation of the gradients. First compute the cost variation for the variation of a single weight

```
from lxmls.deep_learning.rnn import get_rnn_parameter_handlers, get_rnn_loss_range

# Get functions to get and set values of a particular weight of the model
get_parameter, set_parameter = get_rnn_parameter_handlers(
    layer_index=-1,
    row=0,
    column=0
)

# Get batch of data
batch = data.batches('train', batch_size=1)[0]

# Get loss and weight value
current_loss = model.cross_entropy_loss(batch['input'], batch['output'])
current_weight = get_parameter(model.parameters)

# Get range of values of the weight and loss around current parameters values
weight_range, loss_range = get_rnn_loss_range(model, get_parameter, set_parameter, batch)
```

then compute the desired gradient from your implementation

```
# Get the gradient value for that weight
gradients = model.backpropagation(batch['input'], batch['output'])
current_gradient = get_parameter(gradients)
```

and finally call matlplotlib to plot the loss variation versus the gradient

```
%matplotlib inline
import matplotlib.pyplot as plt
# Plot empirical
plt.plot(weight_range, loss_range)
plt.plot(current_weight, current_loss, 'xr')
plt.ylabel('loss value')
plt.xlabel('weight value')
# Plot real
h = plt.plot(
    weight_range,
    current_gradient*(weight_range - current_weight) + current_loss,
    'r--'
)
```

After you have completed the gradients you can run the model in the POS task

```python
import numpy as np
import time

# Hyper-parameters
num_epochs = 20

# Get batch iterators for train and test
train_batches = data.batches('train', batch_size=1)
dev_set = data.batches('dev', batch_size=1)
test_set = data.batches('test', batch_size=1)

# Epoch loop
start = time.time()
for epoch in range(num_epochs):

    # Batch loop
    for batch in train_batches:
        model.update(input=batch['input'], output=batch['output'])

    # Evaluation dev
    is_hit = []
    for batch in dev_set:
        is_hit.extend(model.predict(input=batch['input']) == batch['output'])
    accuracy = 100*np.mean(is_hit)
    print("Epoch %d: dev accuracy %2.2f %%" % (epoch+1, accuracy))

print("Training took %2.2f seconds per epoch" % ((time.time() - start)/num_epochs))

# Evaluation test
is_hit = []
for batch in test_set:
    is_hit.extend(model.predict(input=batch['input']) == batch['output'])
accuracy = 100*np.mean(is_hit)

# Inform user
print("Test accuracy %2.2f %%" % accuracy)
```

3.2 Implementing your own RNN in Pytorch

One of the big advantages of toolkits like Pytorch or Dynet is that creating computation graphs that dynamically change size is very simple. In many other tookits it is directly not possible to use a Python for loop with a variable length to define a computation graph. Again, as in other toolkits we will only need to create the forward pass of the RNN and the gradients will be computed automatically for us.

Exercise 3.2 *As we did with the feed-forward network, we will now implement a Recurrent Neural Network (RNN) in Pytorch. For this complete the log_forward() method in lxmls/deep_learning/pytorch_models/rnn.py.*
Load the RNN model in numpy and Pytorch for comparison

```
# Numpy version
from lxmls.deep_learning.numpy_models.rnn import NumpyRNN
numpy_model = NumpyRNN(
    input_size=data.input_size,
    embedding_size=50,
    hidden_size=20,
    output_size=data.output_size,
    learning_rate=0.1
)

# Pytorch version
from lxmls.deep_learning.pytorch_models.rnn import PytorchRNN
model = PytorchRNN(
    input_size=data.input_size,
    embedding_size=embedding_size,
    hidden_size=hidden_size,
    output_size=data.output_size,
    learning_rate=learning_rate
)
```

To debug your code you can compare the numpy and Pytorch gradients using

```
# Get gradients for both models
batch = data.batches('train', batch_size=1)[0]
gradient_numpy = numpy_model.backpropagation(batch['input'], batch['output'])
gradient = model.backpropagation(batch['input'], batch['output'])
```

and then plotting them with matplotlib

```
%matplotlib inline
import matplotlib.pyplot as plt
# Gradient for  word embeddings in the example
plt.subplot(2,2,1)
plt.imshow(gradient_numpy[0][batch['input'], :], aspect='auto', interpolation='nearest')
plt.colorbar()
plt.subplot(2,2,2)
plt.imshow(gradient[0].numpy()[batch['input'], :], aspect='auto', interpolation='nearest'
    )
plt.colorbar()
# Gradient for  word embeddings in the example
plt.subplot(2,2,3)
plt.imshow(gradient_numpy[1], aspect='auto', interpolation='nearest')
plt.colorbar()
plt.subplot(2,2,4)
plt.imshow(gradient[1].numpy(), aspect='auto', interpolation='nearest')
plt.colorbar()
plt.show()
```

Once you are confident that your implementation is working correctly you can run it on the POS task using the Pytorch code from the Exercise 3.1.

Day 4

Transformers

4.1 Today's assignment

Today's class will demystify the Transformer architecture Vaswani et al. (2017). Transformers have become the dominant model in Natural Language Processing, pushing the performance of many tasks through models like BERT Devlin et al. (2018) and GPT Brown et al. (2020a). In this class, we will review the transformer architecture. The focus of the class is to understand and implement the self-attention mechanism in Pytorch. For this, we will use an annotated version of Karpathy's minGPT code (`https://github.com/karpathy/minGPT`).

4.2 Before Diving into Transformers

Before diving into transformers, we would like to introduce two important details of defining and training transformers.

4.2.1 Positional Encoding

Different from models with recurrence/convolution, positional information is not encoded in Transformers. Therefore, Vaswani et al. (2017) include the positional embedding as part of the input to Transformers. The most popular way to compute the positional embedding (PE) is to use the sinusoidal expressed as

$$PE_{(pos,2i)} = sin(\frac{pos}{10000^{2i/d_{model}}}), \tag{4.1}$$

$$PE_{(pos,2i+1)} = cos(\frac{pos}{10000^{2i/d_{model}}}), \tag{4.2}$$

where pos is the position and i is the dimension.

4.2.2 Tokenization

Transformers operate on sequences of input. In the case of natural language, these sequences could be whole words, single characters, or something in-between. The process of breaking a text string into these smaller pieces is called tokenization. Most transformers use subword tokenization, which operates on the idea that frequently occurring words should occur as single units, while rare words should be split into smaller pieces that occur more frequently. The most widespread subword tokenization algorithm is byte pair encoding (BPE) (Sennrich et al., 2016), which allows *any* string to be represented as a sequence of pieces from a *finite* vocabulary.

Exercise 4.1 *Tokenization*

Let's see how three commonly-used tokenization techniques can be applied in practice.

1. *Run the word-based tokenizer and try out different* `text`*:*

Authors in alphabetical order: Goncalo Melo, Grig Vardayan, Hovhanes Tamoyan, Israfel Salazar, Li Haau-Sing, Roberto Dessi, Venelin Kovachev

```
text = "I travelled to Lisbon in July to attend an NLP summer school"
text.split()
```

2. *Run the character-based tokenizer and try out different different* `text`. *Can you imagine what problems character-level tokenizers pose to NLP models?*

```
text = "I will travel to Lisbon in July..."
tokenized = [c for c in text if c not in [",", ";", ":", "'", "!", "?"]]
print(tokenized)
```

3. *Run the BPE (byte-pair encoding) tokenizer. Can you guess which words will be split subwords and which ones won't?*

```
from lxmls.transformers.bpe import BPETokenizer

tokenizer = BPETokenizer()

sentence = "Your drawing is charmingly anachronistic."
tokenizer.encoder.encode_and_show_work(sentence)
```

4.3 Transformer Architectures

The basic building block of the Transformer combines a large feed-forward network with a multi-head self-attention layer. We have seen feed-forward models in previous days and we will focus here on multi-head self-attention. Refer to previous days for details on the basics of feed-forward models. There are three main varieties of transformer model:

1. Encoders: map a sequence of T observations, e.g. some word or sub-word units $x_1 \cdots x_T$ to a hidden representation of size H, yielding a matrix of embeddings of size (H, T). These contextual embeddings can be combined with a task-specific prediction layer, as in BERT.

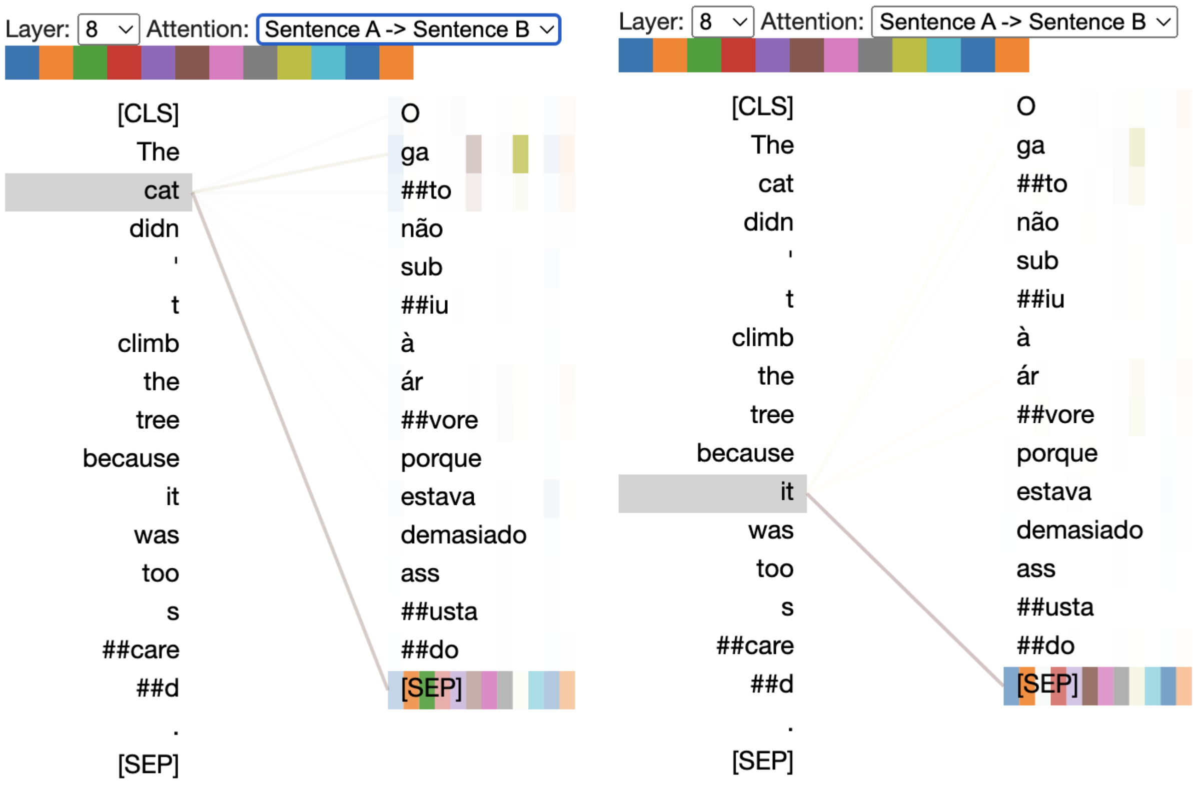
2. Decoders: given a sequence of $T - 1$ observations, e.g. some word or sub-word units $x_{<T} = x_1 \cdots x_{T-1}$, predict the next word x_T. This architecture is the basis of many common large language models, such as GPT.

3. Encoder-Decoders: map a sequence x to another sequence y, which might be of a different length. For this, first x is processed with an Encoder. Then y is processed with a modified Decoder which uses a cross-attention mechanism to compute attention between x and y. Encoder-Decoder models are commonly used for machine translation, as well as in some general-purpose language models such as T5.

Although these models are used for different purposes, the architectures are very similar. In these exercises, we focus first on the Encoder and then explain the other two.

4.4 Attention

Attention mechanisms have become a crucial component in effective sequence modeling across different tasks like machine translation (Bahdanau et al., 2014), and they predate transformers by several years. The general idea of an attention mechanism is that it assigns scores to the hidden states of a model so that it knows where to focus. Figure 4.1 shows how an attention mechanism can decide which source words to focus on when generating each target word. In the left image, we can clearly see that the attention of the token 'cat' is specifically directed towards the token 'gato', which corresponds to its accurate translation. Similarly, in the right image, the word 'it' is attentively aligned with the token 'gato' once again. Transformers Vaswani et al.



Layer: 8 Attention: Sentence A -> Sentence B

Layer: 8 Attention: Sentence A -> Sentence B

Figure 4.1: Encoder-Decoder Attention. Attention distributions were generated with Multilingual BERT (Devlin et al., 2018).

(2017) use several varieties of attention, most famously self-attention, in which scores are produced to model the relationship between all pairs of tokens in a single sequence (see Figure 4.2).

In order to understand why attention mechanisms are such powerful tools, it is helpful to understand the architectural limitations of models like RNNs. In an RNN, the hidden state h_t has a direct connection to the previous hidden state h_{t-1}, but not from all hidden states elsewhere in the sequence. This leads to information loss because all relevant information at time t needs to be put into a single fixed-dimensionality vector – information from several steps in the past may be lost, so these models tend to have a recency bias. This can be a problem in natural language applications because there are often complex relationships that depend on words far away in the sequence, as in the example in Figure 4.3. For a human being, the question of whether the pronoun "it" refers to the cat or the tree is relatively simple due to our prior knowledge about cats and trees. With our understanding of these concepts, we can confidently determine the intended reference of the word "it." For a model, however, discerning the referent of the pronoun "it" – whether it is the cat or the tree – may be more of a challenge because "the tree" occurred more recently. An attention mechanism enables a model to access all hidden states $h_1 \cdots h_T$, not just the most recent.

Now let's explore each building block of the Transformer architecture and examine how the Attention mechanism fits in this.

4.5 Encoder Architecture

In the original paper for transformers (Vaswani et al., 2017), a "sub-layer" is formed by either a Feed-Forward (FF) or Multi-Head Attention MHA (or other named sub $-$ blocks) followed by a sequence of operations. In this section, we begin by explaining the encoder architecture. Note that the notation of "sub-layer" also applies to the following sections.

4.5.1 Simplified Encoder Architecture

The encoder architecture can be succinctly described as stacking a number of N blocks on top of each other combining a FF and MHA sub-blocks. A single block is defined as

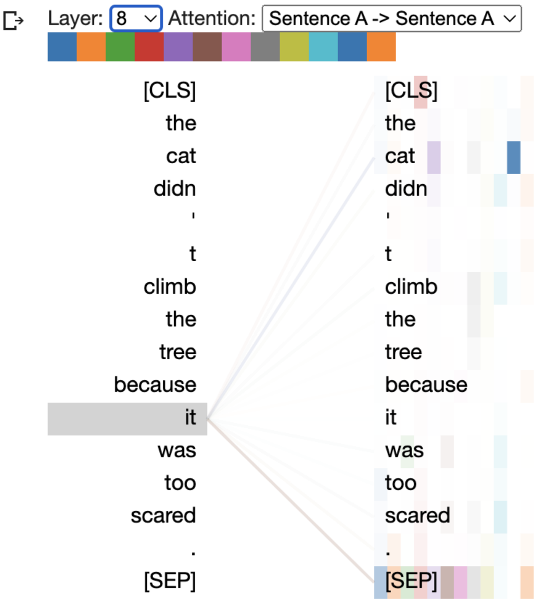

Layer: 8 Attention: Sentence A -> Sentence A

Figure 4.2: Self-Attention

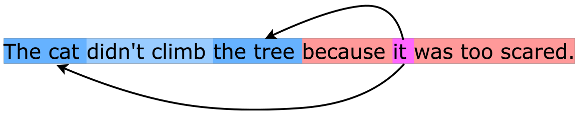


Figure 4.3: An example illustration

$$e^{n+1} = \text{FF}((MHA(e^n)). \tag{4.3}$$

The input to the first block e^0 is the sum of position P and non-contextual embeddings E of the input. Assuming $x_1 \cdots x_T$ is a sequence of integers (indices to a vocabulary of V symbols) we have that

$$e_t^0 = \text{E} \cdot 1_{x_t} + \text{P} \cdot 1_t \quad \text{for} \quad t = 1 \cdots T \tag{4.4}$$

where 1_{x_t} and 1_t are indicators, i.e. one-hot, vectors for the token content (vocabulary symbol) and the token position. $\text{E} \in \mathbb{R}^{H \times V}$ is the non-contextual embedding matrix for each symbol in the vocabulary and $P^{H \times \tau}$ is the position embedding matrix, where $\tau - 1$ is the furthest position supported. See Figure 4.4 for the construction of the input of the first block.

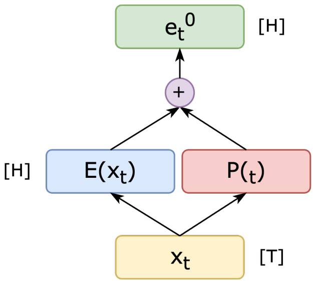


Figure 4.4: Construction of the input of the t-th token in layer e^0

The feed-forward (FF) is given by:

$$FF(z) = W^2 \cdot \text{gelu}(W^1 \cdot z) \tag{4.5}$$

with weight matrices $W^2 \in \mathbb{R}^{H \times 4H}$ and $W^1 \in \mathbb{R}^{4H \times H}$, that expand and contract the hidden dimension H.

Attention consists of three matrices: Query (Q), Key (K), and Value (V). To obtain Q, K, and V, we perform a linear projection of the input using corresponding matrices that are learned during the training process.

The concept of Query, Key, and Value is similar to retrieval systems. When searching for an article on the web using a specific Query, the search engine maps the Query against a set of Keys or titles associated with candidate results in their database. It then presents the best-matched articles or Values to the user. In concrete terms, we can think of it as a weighted modification of a query Q, given some context K and V. There are two cases to consider:

1. Cross-Attention: The query $Q = W^Q \cdot z_q$ is a query from one sequence, and $K = W^K \cdot z_c$ and $V = W^V \cdot z_c$ are the context from another sequence.

2. Self-Attention: The query $Q = W^Q \cdot z$ is a query from one sequence, and $K = W^K \cdot z$ and $V = W^V \cdot z$ are the context from the same sequence.

In Figure 4.5, we present the computation for K, Q, and V in a cross-attention scenario with a given input z_q and a context z_c. Notice that this case is more general, for self-attention, we have $z_q = z_c = z$.

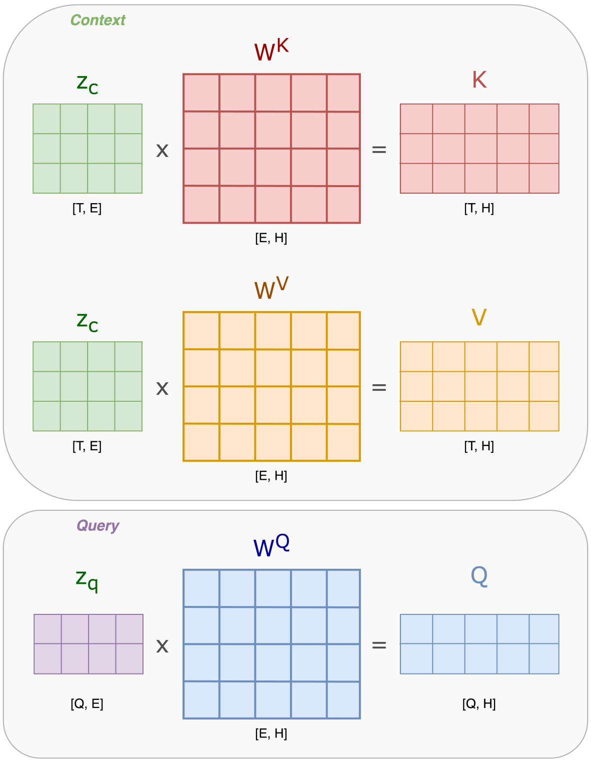


Figure 4.5: Query, Key, and Value projections from a given query z_q, and context z_c.

Then, to perform attention, we follow these steps:

1. Measure Similarity: We calculate the similarity between, Query and Key. This is often done by taking the dot product of the two matrices.

2. Find Maximum Match: We extract the key with the maximum match. This means identifying the Key that has the highest similarity with the Query.

3. Retrieve Value: Once we have the Key with the maximum match, we can retrieve the corresponding Value associated with that Key.

This simple 3-step process is in fact the Attention mechanism. Where we are looking for the best matching tokens for the source sequence in the target sequence. Let's compute the cross-attention score step-by-step. The Key and Query matrices are multiplied by each other and normalized by a constant value $\frac{1}{A} W^Q z_q \left(W^K z_c\right)^T$ or $\frac{1}{A} QK^T$. The A value is usually chosen to be the square root of the dimension of the key matrices. This leads to having more stable gradients.
Then pass the result through a $softmax$ operation. The Softmax normalizes the scores so they're all positive and add up to 1. In other words, extracts a distribution of relative scores from a given token for each token in the input sequence.

After this multiply the output by the Value matrix: $\underset{i\rightarrow}{\text{softmax}}\left(\frac{1}{A}W^Q z\left(W^K z\right)^T\right)\cdot W^V\cdot z$ or simply $V\cdot\underset{i\rightarrow}{\text{softmax}}\left(\frac{1}{A}QK^T\right)$. See Figure 4.6

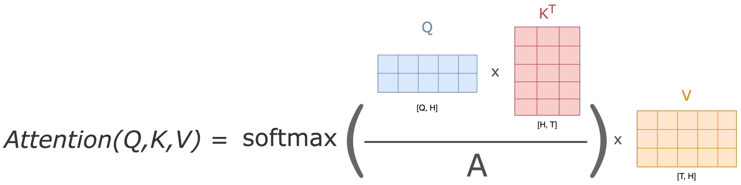


Figure 4.6: The attention mechanism

The Multi-Head Attention (MHA) enhances the Attention layer in two ways: it allows the model to focus on different positions within the input, enabling a better understanding of pronoun references; and allowing projection of input embeddings into distinct representation subspaces, thereby improving overall performance. Finally getting the following mathematical form for the MHA:

$$MHA(z) = W^o\cdot\begin{bmatrix}\underset{i\rightarrow}{\text{softmax}}\left(\frac{1}{A}W_1^Q z\left(W_1^K z\right)^T\right)\cdot W_1^V\cdot z\\ \underset{i\rightarrow}{\text{softmax}}\left(\frac{1}{A}W_2^Q z\left(W_2^K z\right)^T\right)\cdot W_2^V\cdot z\\ \cdots\\ \underset{i\rightarrow}{\text{softmax}}\left(\frac{1}{A}W_D^Q z\left(W_D^K z\right)^T\right)\cdot W_D^V\cdot z\end{bmatrix}$$

where we have D attention heads. Each head contracts the hidden dimension $W^K, W^Q, W^V \in \mathbb{R}^{E\times H}$ into a space of size E which is equal to H/D (this has practical implementation consequences, H and D should be set properly). Outputs of all heads are concatenated and projected again with $W^o \in \mathbb{R}^{H\times H}$. Please note that the dimension of W^V can be different in the general case. The above-mentioned dimensions are correct for the cases when the source and target sequences have equal lengths, which is mostly not the case.

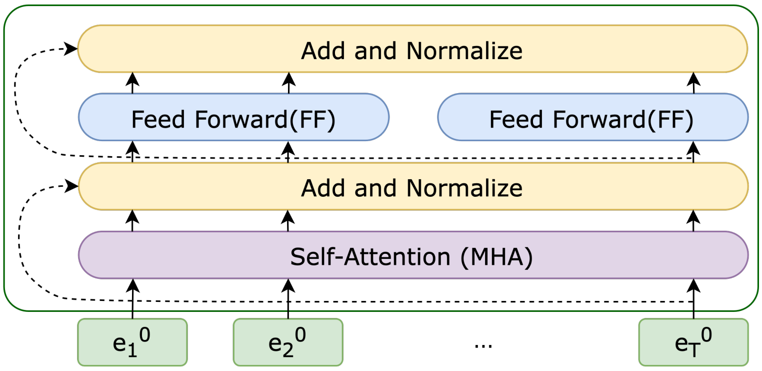


Figure 4.7: Encoder block

4.5.2 Adding Dropout, Residuals, and Layer Normalization

To complete the "sub-layer", a dropout followed by a residual connection and a layer normalization is applied to input z that has been passed through a $\text{sub}-\text{block}$, where a $\text{sub}-\text{block} \in \{\text{FF}, MHA\}$. Together, we can express the function as

$$C(x) = \text{layernorm}(\text{residual}(\text{dropout}(\text{sublayer}(z)), z)) \tag{4.6}$$

where sublayer $\in \{\text{FF}, \text{MHA}\}$.
Specifically, we have

- $\text{dropout}(x) = r * x$ where $r_j \sim \text{Bernoulli}(p)$ and $*$ refers to element-wise multiplication,
- $\text{residual}(x, z) = x + z$, and

Note that the default p is 0.1.

- layernorm$(x) = \frac{x-E(x)}{\sqrt{Var(x)+\epsilon}} * \gamma + \beta$ where $E(x)$ and $Var(x)$ are computed among dimensions other than the dimension for the batch.

After incorporating all of these subcomponents, we obtain the final Encoder block, as illustrated in Figure 4.7

4.6 Decoder Architecture

This is identical to the Encoder architecture with two differences

1. We feed if a partial sequence $x_{<t}$ and take the last output h_{t-1} as the hidden vector for x_t
2. We mask every head of MHA to prevent any value of time p to depend on values of $> p$

The implementation of training realizes 1) by masking input partial sequence $x_{<t}$ and hidden units from the corresponding positions with an attention mask. This attention mask is also applied during inference time.

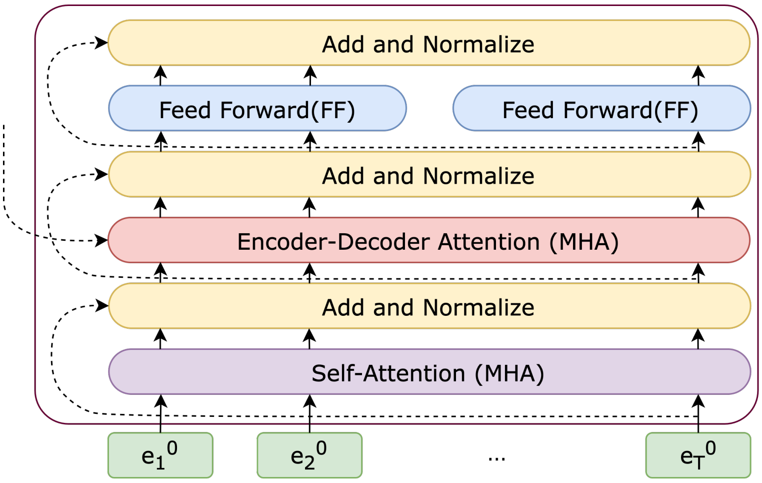


Figure 4.8: Decoder block

Once all these subcomponents are integrated, we achieve the final Decoder block, which is depicted in Figure 4.8.

4.7 Encoder-Decoder Architecture

In the Encoder-Decoder Architecture, the encoder is the same as section 4.5. Therefore, we have the encoded embeddings as $e^N = \text{Encoder}(x)$.

The decoder in the Encoder-Decoder Architecture is a little bit different than section 4.6. Each layer of the decoder will include one more Cross-Multi-Head Attention (CMHA) sub-layer, and the order of sub-layers in a decoder layer will be MHA-CMHA-FF, i.e.

$$d^{m+1} = \text{FF}((CMHA(e^N, MHA(d^m))). \tag{4.7}$$

Specifically, the query matrix is computed from the layer below it, while the key and value matrix are computed from e^N, which can be expressed as

$$\text{CMHA}(e^N, MHA(d^m)) = W^o \cdot \begin{bmatrix} \underset{i\rightarrow}{\text{softmax}}\left(\frac{1}{A} W_1^Q MHA(d^m)\,(W_1^K e^N)^T\right) \cdot W_1^V \cdot e^N \\ \underset{i\rightarrow}{\text{softmax}}\left(\frac{1}{A} W_2^Q MHA(d^m)\,(W_2^K e^N)^T\right) \cdot W_2^V \cdot e^N \\ \dots \\ \underset{i\rightarrow}{\text{softmax}}\left(\frac{1}{A} W_D^Q MHA(d^m)\,(W_D^K e^N)^T\right) \cdot W_D^V \cdot e^N \end{bmatrix}.$$

By combining N blocks of Encoder and Decoder, we obtain the complete view of the Transformer architecture, as illustrated in Figure 4.9.

Note that the default ϵ is $1e-5$. γ and β are learnable affine parameters if we want to learn, otherwise set to 1 and 0

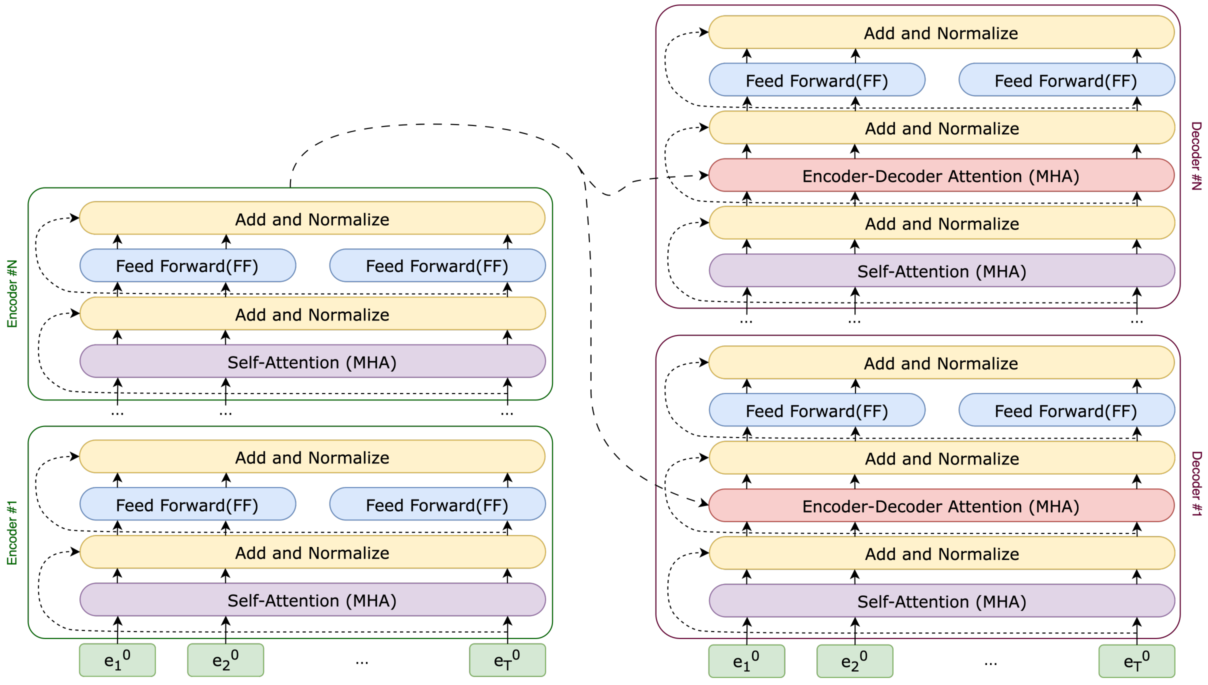


Figure 4.9: Encoder-Decoder Architecture

Exercise 4.2 *Cross Multi-Head Attention & Multi-Head Attention Now let's implement our own Attention mechanism, to that end let's:*

1. *Complete the* `cross_attention` *function. Given two input sequences S_1 and S_2 and the transformation weights W_Q, W_K, and W_V. You need to implement the following:*

   (a) *Calculate the query, key, and value projections using linear transformations.*

   (b) *Compute the attention scores by performing the dot product between the query and key tensors.*

   (c) *Apply softmax activation to the attention scores to obtain the attention weights.*

   (d) *Multiply the attention weights with the value tensor to get the attended values.*

   (e) *Return the attended values.*

```
import torch
import torch.nn.functional as F

def cross_attention(S1, S2, W_Q, W_K, W_V):
    ## Your code here
    return attended_values
```

2. *Complete the* `CausalSelfAttention` *class. First, you should create linear projections* `query_proj`, `key_proj` *and* `value_proj`.

```
import math
import torch.nn as nn

class CausalSelfAttention(nn.Module):

    def __init__(self, config):
        super().__init__()

        # Initialize layers and parameters
        self.hidden_size = config.n_embd
        self.num_heads = config.n_head

        # TODO: Create the linear projections
        self.output_proj = nn.Linear(config.n_embd, config.n_embd)

        self.attn_dropout = nn.Dropout(config.attn_pdrop)
        self.resid_dropout = nn.Dropout(config.resid_pdrop)

        self.register_buffer(
            "bias",
            torch.tril(torch.ones(config.block_size, config.block_size)).view(
                1, 1, config.block_size, config.block_size))
```

Then apply `query_proj`*,* `key_proj` *and* `value_proj` *to split input into query, key, and value tensors:*

```
    def forward(self, x: torch.Tensor) -> torch.Tensor:
        B, T, C = x.size()

        # TODO: Split input into query, key, and value tensors

        # Reshape and transpose tensors for multi-head computation
        query = query.view(B, T, self.num_heads,
                           self.hidden_size // self.num_heads).transpose(1, 2)
        key = key.view(B, T, self.num_heads,
                       self.hidden_size // self.num_heads).transpose(1, 2)
        value = value.view(B, T, self.num_heads,
                           self.hidden_size // self.num_heads).transpose(1, 2)
```

Then compute the attention scores. Note that the shape of scores should be `(B, num_heads, T, T)`

```
        # TODO: Compute attention scores

        # Apply a causal mask to restrict attention to the left in the input
            sequence
        mask = self.bias[:, :, :T, :T]
        scores = scores.masked_fill(mask == 0, float('-inf'))
```

Finally, apply soft-max activation and multiply attention weights with values to get attended values:

```
        # TODO: Apply softmax activation to get attention weights

        # TODO: Multiply attention weights with values to get attended values

        # Transpose and reshape attended values to restore original shape
        attended_values = attended_values.transpose(1, 2).contiguous().view(
            B, T, C)

        # Apply output projection and dropout
        output = self.resid_dropout(self.output_proj(attended_values))

        return output
```

4.8 Training different Transformers

We have introduced different architectures of transformers and will present different variants of transformers that are trained with different objectives, and thus applied to different downstream tasks. We briefly introduce all these transformers and provide references so you can read these papers if you are interested.

4.8.1 Training with Encoders

Encoder models solely utilize the encoder component of a Transformer model. In each step, the attention layers have the ability to consider all the words present in the original sentence. These models are known as 'auto-encoding models' and are recognized for their 'bi-directional' attention mechanism.

The pretraining of these models typically involves manipulating a given sentence, often by obscuring random words and assigning the model the task of identifying or reconstructing the original sentence. This objective is commonly referred to as Masked Language Modeling (MLM), and models pre-trained with this objective are known as Masked Language Models.

For a text sequence $\mathbf{x}$, the BERT model first constructs a corrupted version $\hat{\mathbf{x}}$. Let the masked tokens be $\bar{\mathbf{x}}$. The training objective is to reconstruct $\bar{\mathbf{x}}$ from $\hat{\mathbf{x}}$:

$$max_\theta log P_\theta(\bar{\mathbf{x}}|\hat{\mathbf{x}}) \approx \sum_{t=1}^{T} m_t log P_\theta(x_t|\hat{\mathbf{x}}) = \sum_{t=1}^{T} m_t log \frac{exp(H_\theta(\hat{\mathbf{x}})_t^T e(x_t))}{\sum_{x\prime} exp(H_\theta(\hat{\mathbf{x}})_t^T e(x\prime))} \tag{4.8}$$

Where the $\approx$ indicates that all $\bar{\mathbf{x}}$ elements are independent (seperately reconstructed), m_t indicates weather or not x_t is masked (1-masked, 0-not), H_θ is a Transformer that maps a sequence x of length T and contains infomation about the context on both sides $H_\theta(X) = [H_\theta(x)_1, H_\theta(x)_2, ...H_\theta(x)_T]$, and $e(x)$ denotes the embedding of x.

Encoder models excel in tasks that demand a comprehensive understanding of the entire sentence. These tasks include sentence classification, word classification tasks e.g. named entity recognition, and extractive question answering. Widely used representatives of this model family include BERT Devlin et al. (2018) and RoBERTa Liu et al. (2019).

BERT Devlin et al. (2018) is trained on MLM(Masked Language Modeling) and NSP(Next Sentence Prediction) objectives. In the **MLM** objective, 15% of the input tokens are selected. Among these selected tokens:

- 80% of the time, the mask token is inserted in place of the original token.
- 10% of the time, a random token is inserted in place of the original token.
- 10% of the time, the original token remains unchanged.

The **NSP** objective is applied to pairs of sentences, A and B, taken from the training set. In 50% of the cases, sentence B directly follows sentence A in the input, while in other cases, the pairs are randomly selected. The objective is to perform binary classification and predict whether sentence B follows sentence A or not.

RoBERTa Liu et al. (2019) builds upon BERT's pre-training by addressing its perceived undertraining. It introduces the following modifications:

- Longer and larger-scale training: RoBERTa trains the model for an extended period using larger batches, more data, and longer sequences.
- Removal of NSP objective: The next sentence prediction (NSP) objective, present in BERT, is eliminated in RoBERTa.
- Dynamic masking: RoBERTa employs dynamic masking by duplicating the training data ten times and applying different mask patterns to each sample. This contrasts with BERT's static masking, where a fixed mask is used for each sample.

These modifications aim to enhance RoBERTa's pre-training performance and overall language understanding capabilities.

4.8.2 Training with Decoders

Decoder-only or Autoregressive modeling performs pretraining by maximizing the likelihood under the forward autoregressive factorization:

$$max_\theta log P_\theta(\mathbf{x}) = \sum_{t=1}^{T} log P_\theta(x_t|\mathbf{x}_{<t}) = \sum_{t=1}^{T} log \frac{exp(h_\theta(\mathbf{x}_{1:t-1})^T e(x_t))}{\sum_{x\prime} exp(h_\theta(\mathbf{x}_{1:t-1})^T e(x\prime))}$$

Where the $h_\theta(\mathbf{x}_{1:t-1})$ is the context representation produced by the neural model, and it contains information (conditioned) about the tokens up to position t, and $e(x_t)$ is the embedding of a token x_t.

A Decoder-only or Autoregressive model operates differently compared to Encoder-only or Autoencoder model by focusing on density estimation rather than reconstructing corrupted input. These models aim to estimate probability distributions, which limits their ability to capture bidirectional context. As a result, they are restricted to uni-directional processing.

GPTs. The class of GPTs is a series of pre-trained decoder-only transformers. Models are pre-trained to perform the next token prediction with the Cross-Entropy criterion. Since the release of GPT-1 (Radford et al., 2018), GPTs are being trained with more parameters and training data, with GPT-2 (Radford et al., 2019), GPT-3 (Brown et al., 2020b), InstructGPT (Ouyang et al., 2022) and GPT-4 (OpenAI, 2023) being subsequently released. Note that since the release of InstructGPT (Ouyang et al., 2022), this class of language models has been trained with reinforcement learning with human feedback (Bai et al., 2022), which enables the model to manifest interesting behaviors that you see when using ChatGPT.

4.8.3 Training with Encoder-Decoder

To leverage the strengths of both Encoder-only and Decoder-only models, the concept of Encoder-Decoder models was introduced. While previous methods effectively captured bidirectional information for text generation, they had certain limitations in terms of contextual token representations. One common approach involved concatenating the left-to-right and right-to-left representations Peters et al. (2018).

Encoder-Decoder models aim to overcome these limitations by combining the advantages of both approaches. These models can effectively capture bidirectional information while maintaining robust contextual representations for each token. By leveraging the strengths of both encoders and decoders, Encoder-Decoder models offer enhanced capabilities for various natural language processing tasks, including text generation.

Commonly employed members of this model family include BART Lewis et al. (2020) and T5 Raffel et al. (2020), which have gained significant popularity in the field.

BART. BART Lewis et al. (2020) an encoder-decoder model pre-trained on five tasks injecting noises into the input text: i) token masking (same as BERT Devlin et al. (2018)), ii) token deletion, iii) text infilling by replacing sampled input spans with single masks, iv) sentence permutation, and v) document rotation. The powerful pre-trained denoising autoencoder is commonly used in generation tasks.

T5. T5 (Raffel et al., 2020) utilizes a text-to-text methodology. In this approach, various tasks like translation, question answering, and classification are transformed into a unified format. The model is provided with input text and trained to generate the corresponding output text. To achieve this, a task-specific prefix(instruction) is added to the input sequence, and the model is pre-trained to produce outputs specific to each task.

Exercise 4.3 *Training Transformers*

Let's delve into practical training of a Transformer model to gain hands-on experience. Our initial step will involve creating a Decoder-based model (GPT-2 Radford et al. (2019)) and training it on a modest dataset. Once this is completed, we can visualize the attention mechanism before and after the training process. Finally, we can compare the performance of the trained model during inference with that of the smaller trained model.

Exercise 4.4 *Training a Weather Prediction Model Using Autoregressive Transformer.*

In this exercise, we will be working with a dummy weather dataset comprising sequences of weather observations and their corresponding states. The objective is to train a compact autoregressive transformer model that can predict the weather state based on previous observations. Let's begin by importing the required modules and classes, and setting a seed value to ensure consistent output every time we run the file.

```
import random
import time

import numpy as np
import torch
from torch.utils.data.dataloader import DataLoader

random.seed(42)

from lxmls.transformers.bpe import BPETokenizer
from lxmls.transformers.dataset import WeatherDataset
from lxmls.transformers.model import GPT
from lxmls.transformers.trainer import Trainer
from lxmls.transformers.utils import set_seed
```

To begin, we initialize the training dataset, which comprises sequences of weather observations along with their corresponding states. These sequences are transformed into indices and then concatenated to create the input and output sequences for the transformer model. For more information on this, refer to: ”lxmls/transformers/dataset.py”.

```
fixed_proba = {}
fixed_proba["initial"] = [0.5, 0.3, 0.2]
fixed_proba["transition"] = [[0.5, 0.5, 0], [0, 0.5, 0.5], [0.5, 0, 0.5]]
fixed_proba["emission"] = [
    [0.5, 0, 0.2, 0, 0.3],
    [0, 0.5, 0.4, 0, 0.1],
    [0, 0, 0.1, 0.5, 0.4],
]
```

Let's print an example instance of the dataset.

```
train_dataset = WeatherDataset("train", proba=fixed_proba)
test_dataset = WeatherDataset("test", proba=train_dataset.proba)
x, y = train_dataset[0]

print("Sampling from the dataset:")
print(f"Input: {train_dataset.decode_obs(x.tolist()[:6])}")
print(f"Labels: {train_dataset.decode_st(y.tolist()[5:])}")
print("-" * 50)
print("Tokenized sequences:")
print(f"Input: {x.tolist()}")
print(f"Labels: {y.tolist()}")
```

Moving forward, we construct a model using the default configuration for the GPT model, which encompasses parameters defining the model's size and structure. In this case, we employ a compact variant known as GPT Nano.

```
model_config = GPT.get_default_config()
model_config.model_type = "gpt-nano"
model_config.vocab_size = train_dataset.get_vocab_size()
model_config.block_size = train_dataset.get_block_size()
model = GPT(model_config)

print(model_config)
```

To facilitate the training of our model, we instantiate a Trainer object. The Trainer manages various aspects of the training process, such as configuring the learning rate, specifying the maximum number of iterations, and determining the number of workers for data loading. We initialize the Trainer with the model, training dataset, and validation dataset.

```
train_config = Trainer.get_default_config()
train_config.learning_rate = (
    5e-4  # The model we're using is so small that we can go a bit faster
)
train_config.max_iters = 2000
train_config.num_workers = 0
train_config.device = "mps"
trainer = Trainer(train_config, model, train_dataset)

print(train_config)
```

With all these components in position, we are fully prepared to train our model using the weather dataset and leverage the acquired patterns to generate predictions.

```
def batch_end_callback(trainer):
    if trainer.iter_num % 100 == 0:
        print(
            f"iter_dt {trainer.iter_dt * 1000:.2f}ms; iter {trainer.iter_num}: train loss
                 {trainer.loss.item():.5f}"
        )


trainer.set_callback("on_batch_end", batch_end_callback)

start_time = time.time()
trainer.run()
end_time = time.time()
elapsed_time = end_time - start_time

# Print the training time
print("Training time: {:.2f} seconds".format(elapsed_time))
```

Now, let's visualize the attention heads of our trained model. To do this, we need to install the HF transformers and bertviz packages. You can install them by running the following command: "pip install transformers bertviz".

```
from transformers import BertTokenizer, BertModel
from bertviz import head_view

# Define a sample input text
text = "I will go for a run and will jump into a lake."

# Instantiate the BERT tokenizer and model
tokenizer = BertTokenizer.from_pretrained("bert-base-uncased")
model = BertModel.from_pretrained("bert-base-uncased")

# Tokenize the input text
tokens = tokenizer.tokenize(text)

# Convert tokens to token IDs
token_ids = tokenizer.convert_tokens_to_ids(tokens)

# Create attention mask
attention_mask = [1] * len(token_ids)

# Convert token IDs and attention mask to tensors
input_ids = torch.tensor([token_ids])
attention_mask = torch.tensor([attention_mask])

# Generate the transformer output
outputs = model(input_ids, attention_mask=attention_mask, output_attentions=True)

head_view(outputs.attentions, tokens=tokens)
```

Exercise 4.5 *Promoting a GPT-2 model Now that we have trained our own Transformer model and gained some understanding of its functioning, let's explore a larger pre-trained model that has been trained on a vast amount of natural data. By doing so, we can experience generating output that closely resembles human-like responses.*

```
model_type = "gpt2"
device = "mps"

model = GPT.from_pretrained(model_type)

# move model to the device(GPU if available)
# set to eval mode to avoid gradient accumulation
model.to(device)
model.eval()

# Random prompt, uses pooling
for i in range(5):
    set_seed(42)
    model.prompt("Alan Turing, the", 50, 3)

# Deterministic prompt, does NOT use pooling
for i in range(5):
    model.prompt_topK("Alan Turing, the", 50, 3)
```

Get ready to embark on an intriguing journey by playing with the input prompt and witnessing the fascinating and captivating outputs that await you!

Day 5

Multimodal Models

5.1 Today's assignment

Today's class delves into multimodal architectures, focusing on how models learn the relationships between different types of data, including images and text. More specifically, we will explore **CLIP (Contrastive Language-Image Pre-Training)** Radford et al. (2021): a foundational model that learns visual concepts from natural language supervision through a shared embedding space. CLIP's shared embedding space has heavily influenced modern large-scale multimodal systems, stemming from its initial application to models like DALL-E. As a means of encoding textual prompts and conditioning image synthesis, it enables text-driven image generation and manipulation, while also facilitating the ranking of generative outputs.

On that note, the primary focus of this class is to complete this lab's notebook (adapted from a HuggingFace PyTorch implementation of CLIP) sustaining the following milestones: (i) understand CLIP's dual-encoder architecture and contrastive learning, (ii) encode images and captions into the shared embedding space of CLIP, (iii) implement a zero-shot rescoring pipeline by sampling candidate sentences, computing CLIP similarity scores, and selecting the best match, and (iv) visualizing resulting image-text distances of a Gemma **Vision-Language Model (VLM)** and how cross-modal tokenization and convolution play a role.

By the end of this session, you'll have hands-on experience with the core mechanics that make CLIP and many other multimodal systems so powerful.

5.2 Before Diving into Multimodal Models

Before building our model, it is essential first to understand the core concepts that make multimodal learning possible.

5.2.1 Image Tokenization: From Pixels to Patches

Transformers operate on sequences of inputs. In the case of natural language, as we have seen in the previous day, this involves breaking text into smaller pieces through tokenization. A similar concept is required for images: we partition a grid of pixels into a sequence of patches and feed them into a Vision Transformer (ViT) Dosovitskiy et al. (2021). This process, visually demonstrated in Figure 5.1, works through the following sequence of operations:

1. **Patch Extraction and Flatenning:** With an input image $x \in \mathbb{R}^{H \times W \times C}$ (where H and W are the height and width of the image in pixels, respectively, and C is the number of color channels per pixel), split it into a sequence of flattened non-overlapping patches ($x_p \in \mathbb{R}^{N \times (P^2 \cdot C)}$), where:

$$N = \frac{H}{P} \times \frac{W}{P} \tag{5.1}$$

   is the resulting number of patches, each of size $P \times P$ (e.g., 16×16 pixels), and where each flattened patch is a one-dimensional vector x_p, where:

$$x_p = \left[p_{1,1,1},\, p_{1,1,2},\, p_{1,1,3},\, p_{1,2,1},\, p_{1,2,2},\, p_{1,2,3},\, \ldots\right]^\top \in \mathbb{R}^{P^2 \cdot C} \tag{5.2}$$

   These are then treated as ”tokens”, similarly to word tokens of standard transformers;

2. **Linear Projection to Embeddings:** Apply a learnable linear layer ($E \in \mathbb{R}^{(P^2 \cdot C) \times D}$) to each flattened patch to project each patch into a D-dimensional embedding space, such that:

$$x_p^{(i)} = x_p^{(i)} E \quad \in \mathbb{R}^D \tag{5.3}$$

3. **Class Token and Positional Embeddings:** Prepend a learnable class embedding ($z_0^0 = x_{class} \in \mathbb{R}^D$), whose final state ($z_L^0$) will represent the entire image, and, just like text-based transformers, add a learnable one-dimensional positional embedding ($E_{pos} \in \mathbb{R}^{(N+1) \times D}$) to retain spatial information about where each patch was located in the original image, such that:

$$z^0 = [x_{class}; x_p^{(1)} E; x_p^{(2)} E; ...; x_p^{(N)} E] + E_{pos} \tag{5.4}$$

4. **Transformer Encoder:** Feed the previous sequence into a standard transformer encoder of L layers, enabling the self-attention mechanism to weigh the importance of different parts of the image relative to one another.

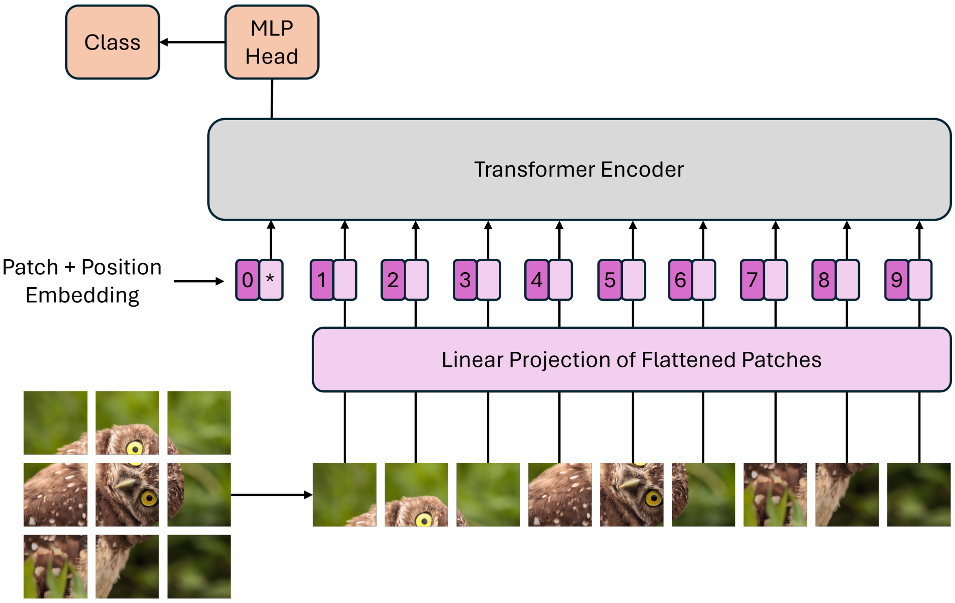

Class
MLP Head
Transformer Encoder
Patch + Position Embedding
0
*
1
2
3
4
5
6
7
8
9
Linear Projection of Flattened Patches

Figure 5.1: An illustration of the image tokenization process used in ViT. An image is split into fixed-size patches, which are then linearly projected and combined with positional embeddings to form a sequence of input vectors for the transformer encoder.

5.2.2 Contrastive Learning

Contrastive learning, as the name suggests, is a training technique where contrastive data samples are evaluated against each other as a means of training a model to learn common and distinct attributes among samples of different classes (e.g., distinguishing a dog from a cat) Chen et al. (2020). In doing so, this technique tunes a model in such a way that allows it to encode embeddings where similar classes are close to one another in the embedding space, as opposed to distinct classes, which have a significant enough distance that allows the model to distinguish them.

More specifically, data samples are often partitioned into three distinct categories: (i) anchors, representing a base class for contrast, (ii) positives, with an equal class to that of the anchor, and (iii) negatives, representing a distinct class to that of the anchor.

On that note, an encoder is trained via contrastive learning such that embeddings of anchor and positive samples are close together, in the embedding space, as opposed to negative samples, which should be far apart. This is done so by tuning the encoder through a maximization or minimization of distance functions (e.g., Euclidean distance or cosine similarity) applied to the anchor and multiple positive samples, and the anchor and multiple negative samples, respectively, as illustrated in Figure 5.2, and described through the following formalisms:

1. **Encoder and Embedding Definition:** Given an input sample I and an encoder θ, the latter produces a d-dimensional embedding from the former:

$$f = \theta(I) \in \mathbb{R}^d \tag{5.5}$$

2. **Anchor, Positive, and Negative Samples Definition:**
   - *Anchor* $I_i^a, i = 1, \ldots, K$: Base samples;
   - *Positive* $I_i^+, i = 1, \ldots, K$: Samples of the same class or augmented views of the anchors;
   - *Negative* $I_j^-, j = 1, \ldots, K$: Samples from different classes.

3. **Embedding Normalization:** For the proper application of a distance function, each embedding must be normalized to unit norm:

$$\hat{f} = \frac{f}{\|f\|} \tag{5.6}$$

4. **Distance Function:** Compute the similarity measure via the distance function (cosine similarity is equivalent to the dot product):

$$\delta(u, v) = u^\top v \tag{5.7}$$

5. **Contrastive Loss:** Provided an anchor I_i^a, with a corresponding positive I_i^+, and negatives $\{I_{i,j}^-\}_{j=1}^K$, the loss for the anchor I_i^a, where $\tau > 0$ is a temperature hyperparameter, is:

$$\mathcal{L}_i = -\log \frac{\exp(\delta(\hat{f}_i, \hat{f}_i^+)/\tau)}{\exp(\delta(\hat{f}_i, \hat{f}_i^+)/\tau) + \sum_{j=1}^K \exp(\delta(\hat{f}_i, \hat{f}_{i,j}^-)/\tau)} \tag{5.8}$$

6. **Overall Loss:** For a batch of N anchors, the overall loss is:

$$\mathcal{L} = \frac{1}{N} \sum_{i=1}^N \mathcal{L}_i \tag{5.9}$$

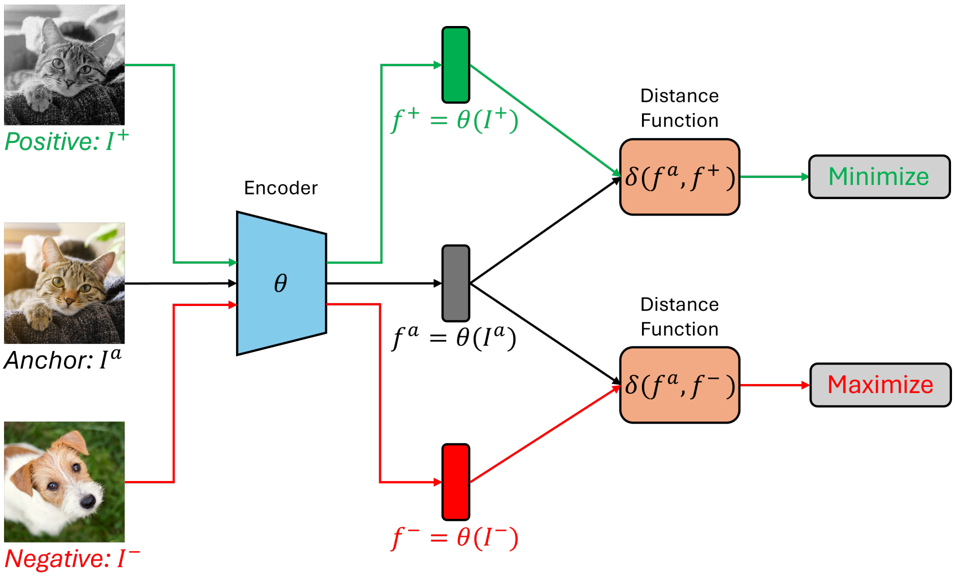


Figure 5.2: A high-level illustration of the contrastive learning framework, where an encoder is trained to minimize and maximize the distance function between an anchor and its contrastive positive and negative samples, respectively. Here, the positive sample is a transformation of the original image.

Finally, it is worth mentioning the role of data augmentation in such frameworks. Positive samples are either images containing data that fits within the same class as the anchor or augmentations of the anchor (i.e., multiple patches and transformations), ranging from color jittering and rotation to noise injection and random affine transformations. This is done to ensure that the model does not conform to unwanted patterns (e.g., pixel-level or rotation-specific artifacts), but instead learns the core features that characterize these classes (e.g., shapes, contours, textures, and spatial configurations), which in turn allows for proper generalization. After training, the resulting encoder should be capable of accurately distinguishing between similar and dissimilar classes, as illustrated in Figure 5.3.

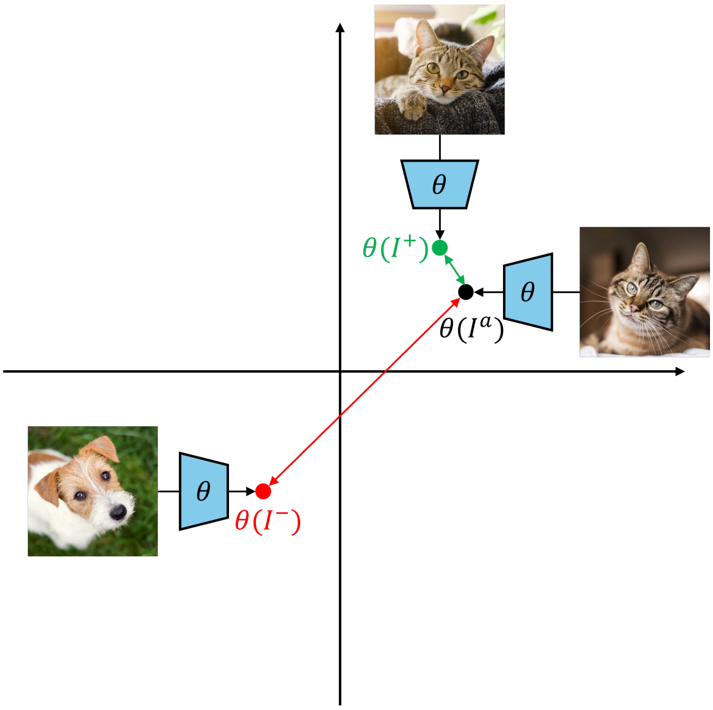


Figure 5.3: An illustration of the embedding space of a properly tuned encoder that sets distinct classes far apart, while shortening the distance of similar samples as per contrastive learning. Here, the black data point serves as the anchor, the green data point as the positive sample, and the red data point as the negative sample. Note the difference in distance from the anchor to the positive and negative samples.

5.3 Multimodal Architectures

The core building block of modern multimodal systems often involves combining powerful pre-trained unimodal models. Unlike standard Transformers, which operate on a single sequence of tokens, multimodal models must compare fundamentally different data structures (e.g., image patches and text tokens). The solution is to create a "common language" known as the shared embedding space. This is a high-dimensional vector space where both images and text can be represented. As such, the goal is to train encoders that map semantically similar image-text pairs close to each other in this space. There are three main approaches:

1. **Dual Encoders:** This architecture, popularized by CLIP, uses two separate encoders—one for each modality (a ViT for images and a standard Transformer for language). The encoders are trained jointly with the contrastive learning to align their output embedding spaces. This design is highly efficient for retrieval tasks.

2. **Fusion Encoders:** These models use a single Transformer encoder that processes the combined inputs from multiple modalities. This enables "cross-attention" between image patches and text tokens at a much deeper level, allowing for more complex reasoning. Models like ViLT and VisualBERT follow this pattern.

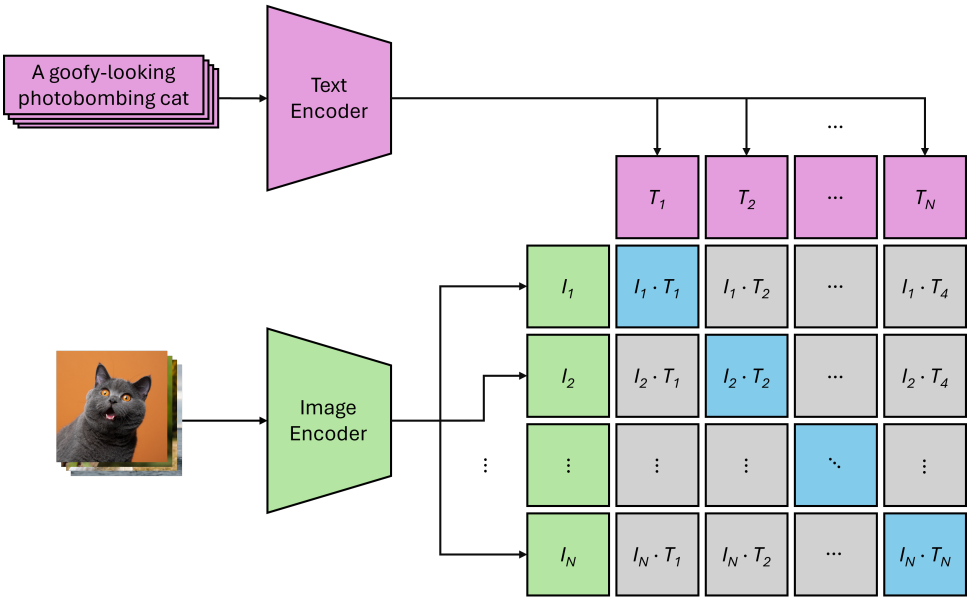
3. **Encoder-Decoders:** These models are used for generative tasks. An encoder processes one modality (e.g., an image), and a decoder uses that representation to generate output in another modality (e.g., a text caption). Models like BLIP and Flamingo are powerful examples used for image captioning and visual question answering.

Even though these models are used for different purposes, their architectures are very similar. Today's exercises, however, will focus on the dual encoder architecture, as it provides the clearest illustration of cross-modal alignment.

5.3.1 CLIP:

As previously detailed, CLIP resorts to a dual encoder architecture to encode images and text in a shared embedding space through a ViT and a standard Transformer, respectively Radford et al. (2021). The idea is to train this model—via contrastive learning—in such a way that maximizes the similarity of the encoded image-text pairs, while minimizing the similarity of disparate encoded image-text pairs (i.e., the embedding of the sentence "a gray british shorthair cat" should be similar to the embedding of an image of a cat, as opposed to that of a dog). In doing so, such allows the establishment of relationships between image-text pairs that serve a myriad of purposes, ranging from image classification to more sophisticated integrations in generative scenarios. One interesting aspect about CLIP is its capability of generalizing to tasks it was not trained on (i.e., zero-shot classification)—a surprising example being CLIP's performance matching the accuracy-level of the famous ResNet-50 without actually training on the data which ResNet-50 trained on.

More specifically, focusing on the base case for image classification, CLIP takes a given number of text embeddings, via a standard Transformer—usually text labels scraped from large datasets of image captions from the web—, followed by multiple image embeddings, via a ViT, matching each of the previous labels. Next, the dot product (i.e., cosine similarity) is applied to each possible image-text embedding pair. Thus, the model is iteratively trained in such a way that matching image-text embeddings stay close together in the embedding space and dissimilar pairs far apart, as per contrastive learning, as illustrated in Figure 5.4.



	T_1	T_2	...	T_N
I_1	$I_1 \cdot T_1$	$I_1 \cdot T_2$	...	$I_1 \cdot T_4$
I_2	$I_2 \cdot T_1$	$I_2 \cdot T_2$	...	$I_2 \cdot T_4$
⋮	⋮	⋮	⋱	⋮
I_N	$I_N \cdot T_1$	$I_N \cdot T_2$	...	$I_N \cdot T_N$

Figure 5.4: An illustration of CLIP's dual encoder architecture. Images and texts are embedded and individually paired with one another via a distance function to rank for similarity. The model is then iteratively tuned to achieve proper distance values and enable accurate classification (the values of the matrix's diagonal should be minimized and the remaining cells maximized).

This last step is done symmetrically—by performing a loss through both ways (i.e., text-to-image and image-to-text) which is later averaged into one component—to ensure proper alignment between both modalities. Since CLIP learns broad vision–language alignments from large-scale web data, it can be applied in zero-shot fashion to downstream tasks without further fine-tuning, simply by filling out a default prompt of the form "a photo of a {class}", where '{class}' is replaced by the label whose text embedding is most similar to the embedding of the image (see Figure 5.5).

Formally speaking, CLIP uses a dual-encoder architecture to map images and text into a shared d-dimensional embedding space, and trains via a symmetric contrastive objective as per the following formalisms:

1. **Encoders and Embeddings Definition:** Given an image domain I and a text domain T, where $f_i = \theta(I_i)$

and $g_i = \phi(T_i)$, we have:

$$\theta : I \to \mathbb{R}^d, \quad \phi : T \to \mathbb{R}^d \tag{5.10}$$

2. **Embedding Normalization:** Each embedding must be normalized to unit form for proper application of the distance function:

$$\hat{f}_i = \frac{f_i}{\|f_i\|}, \quad \hat{g}_i = \frac{g_i}{\|g_i\|} \tag{5.11}$$

3. **Distance Function:** Compute the distance between each image-text embedding pair via cosine similarity:

$$\delta_{ij} = \hat{f}_i \cdot \hat{g}_j \tag{5.12}$$

4. **Symmetric Contrastive Loss:** For a batch of N matched image-text pairs (I_i, T_i), we have a loss function for each direction (i.e., image-to-text and text-to-image), together with the total loss:

$$\mathcal{L}_{\text{I2T}} = -\frac{1}{N}\sum_{i=1}^{N} \log \frac{\exp(\delta_{ii})}{\sum_{j=1}^{N} \exp(\delta_{ij})}, \tag{5.13}$$

$$\mathcal{L}_{\text{T2I}} = -\frac{1}{N}\sum_{j=1}^{N} \log \frac{\exp(\delta_{jj})}{\sum_{i=1}^{N} \exp(\delta_{ij})}, \tag{5.14}$$

$$\mathcal{L} = \frac{1}{2}(\mathcal{L}_{\text{I2T}} + \mathcal{L}_{\text{T2I}}) \tag{5.15}$$

5. **Zero-Shot Classification:** Given an image I and a set of K candidate class names c_k, form prompts $T_k =$ "A photo of a c_k", normalize each prompt to unit norm, and compute the probability of class k matching image I:

$$p(y = k \mid I) = \frac{\exp(\hat{f} \cdot \hat{g}_k)}{\sum_{j=1}^{K} \exp(\hat{f} \cdot \hat{g}_j)} \tag{5.16}$$

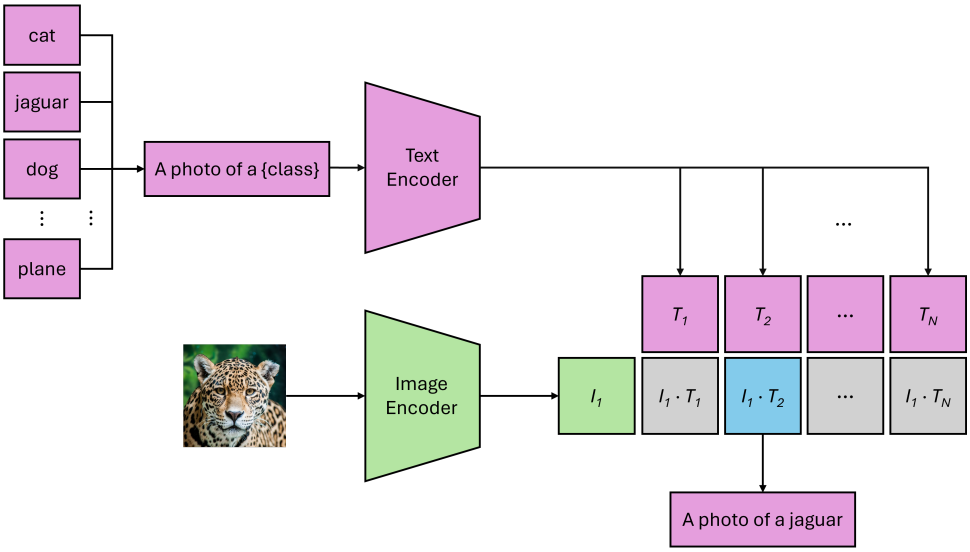


Figure 5.5: An illustration of CLIP's zero-shot classification. After training, CLIP is capable of properly classifying an image provided the properly tuned weights that result from the pre-training. Similarity measures are computed between the image and each possible text pair and the shortest distance denotes the right class.

Exercise 5.1 *Let us now apply the previous principles in practice. You will be implementing multiple functions—each serving a different purpose—covering the processing, embedding, and classification of image-text pairs. In essence, you will be feeding a ViT and a standard Transformer with the intended data, which will allow for image-text pair classification following CLIP's approach. To that end, this exercise will guide you through the completion of the corresponding yet-to-be-completed notebook cells present in* `lxmls/labs/notebooks/multimodal/clip_exercise.ipynb`*:*

1. ***Basic setup:*** *Run the first two cells of the notebook to import all necessary dependencies for proper execution of all code cells.*

2. ***Data and model setup:*** *Run all subsequent cells until Cell 19 (inclusive). This will enable you to automatically execute the following steps:*

    - *Prepare the ViT's input (i.e., loading an example image of two cats);*
    - *Initialize the GPT model from the previous day for image labelling;*
    - *Prepare a string to use as a default for image classification (this will later append individual GPT-generated labels);*
    - *Load and prepare a CLIP model;*
    - *Convert the example image using the image processor;*
    - *Load a tokenizer and tokenize the previously defined image labels;*
    - *Embed the image labels and visualize the outcome of splitting the example image into fixed size patches;*
    - *Store the learned weights for linearly projecting the image.*

3. ***Prepare the image for projection to the embedding space:*** *You will now implement a function whose purpose is to project the image, in its entirety, to the embedding space. For that to happen, define the following steps in sequence:*

    - *Break the image into multiple patches of fixed size (patch_size, patch_size) (Hint: Take a peek at Cell 18. How can we declare patches?);*
    - *Apply a learnable linear layer to each patch (Hint: Take a peek at Cell 19 and Eq. 5.3);*
    - *Assign each linearly projected patch embedding to the* `manual_patch_embeds` *variable.*

```
def get_patch_embeddings(pixel_values, filter_weights):
"""using a similar code as the visualization above, project the image patches in the embedding space"""
    assert len(pixel_values) == 1 # we do it for a single image for the example
    manual_patch_embeds = torch.zeros(num_patches, num_patches, filter_weights.shape[0], device=model.device)

    # YOUR SOLUTION HERE

    return manual_patch_embeds
```

4. ***Flatten the embeddings and check for correctness:*** *Run all subsequent cells until Cell 22 (inclusive). This will enable you to automatically execute the following steps:*

    - *Retrieve the linearly projected patch embeddings, as per the function you've just defined, and flatten it (notice the difference in shape before and after flattening);*
    - *Compare your results with the original implementation to assure correctness (this will allow you to test if you implemented the previous function as intended).*

5. ***Prepare the class token and positional embeddings:*** *You will now implement a function that serves the following purposes:*

    - *Prepend a learnable class embedding to the previously defined patch embeddings (already implemented for convenience)*
    - *Add a learnable positional embedding to retain information about each patch's location in the original image (Hint: Take a peek at Eq. 5.4, look through the possible method calls of the model, and respective arguments)*

```python
def get_add_cls_and_position(manual_patch_embeds):
    """The only thing left to do is to add class and position embedding"""
    class_embeds = model.vision_model.embeddings.class_embedding.expand(batch_size,
        1, -1)
    hidden_states = torch.cat([class_embeds, manual_patch_embeds], dim=1)

    # YOUR SOLUTION HERE

    hidden_states = model.vision_model.pre_layrnorm(hidden_states)
    return hidden_states
```

6. ***Conclude the image embedding and project the image-text pairs:*** *Run all subsequent cells until Cell 27 (inclusive). This will enable you to automatically execute the following steps:*

   - *Retrieve the class and positional embeddings in conjunction with the patch embeddings;*
   - *Feed the previously declared embeddings to the ViT;*
   - *Extract the class token representation;*
   - *Apply the final normalization to obtain the image embedding;*
   - *Project both the text and image embeddings to the shared embedding space (see Figure 5.4 for a visual explanation)*

7. ***Define the distance function:*** *You will now implement a function whose purpose is extracting the similarities between all image-text embedding pairs. To accomplish that, define the following steps in sequence:*

   - *Normalize each embedding (Hint: Take a peek at Eq. 5.11);*
   - *Apply the cosine similarity to the embeddings (Hint: Take a peek at Eq. 5.12)*

```python
def get_sim(text_embedding, image_embedding):

    # YOUR SOLUTION HERE

    return similarities
```

8. ***Retrieve the similarities:*** *Run Cell 29 to retrieve the similarities as per the function you've just defined.*

9. ***Define a rerank function:*** *You will now implement a function whose purpose is reranking the retrieved similarities between all image-text embedding pairs, in descending order, such that the class with highest probability stays on top:*

```python
def rerank(similarities):
    """All is left to do is sort the similarities to rerank the text"""

    # YOUR SOLUTION HERE

    return ranking
```

10. ***Analyze the results:*** *Run all remaining cells. This will enable you to automatically execute the following steps:*

    - *Apply the reranking function to the similarity scores;*
    - *Print the resulting similarities in descending order, guaranteeing that the classification with the highest probability stays on top (see Figure 5.5 for a visual description).*

    What can you conclude by analyzing the results? Does your implementation work as intended?

5.4 VLM: Vision-Language Models

5.4.1 From Retrieval to Generative Vision-Language Models

The next step in multimodality is to go beyond retrieval-based alignment (like in CLIP) to **generative vision language models (VLMs)** that can produce novel text conditioned on images. Instead of merely ranking or classifying images via a shared embedding space, generative VLMs take an image (and optionally text) as input and output descriptive or explanatory text. This enables rich tasks such as image captioning, visual question answering, and even multimodal dialogue.

A common design pattern in these models is to use a pre-trained image encoder to transform an image into a sequence of vector embeddings, then feed those *visual tokens* into a language model that generates text. Intuitively, the image is 'translated' into the same token space as the text, allowing a transformer to handle both modalities jointly. Unlike dual encoders that map each modality to a separate embedding space (e.g., CLIP), generative VLMs typically integrate both within one model, often an encoder-decoder or a decoder-only transformer with modified attention. This allows the model to produce natural language grounded in visual context, rather than just comparing embeddings.

5.4.2 Gemma 3

A recent state-of-the-art VLM is **Gemma 3** (Kamath et al., 2025), released by Google in 2025. It builds upon the Gemma language model family and extends it to handle both text and images using a unified architecture. Gemma 3 comes in sizes up to 27B parameters and is capable of zero-shot generation, captioning, and multimodal dialogue.

At its core, Gemma 3 uses a vision encoder based on **SigLIP** Zhai et al. (2023), which improves upon CLIP's contrastive training by using a pairwise sigmoid loss. In SigLIP, each image–text pair is treated as an independent binary classification task, avoiding the need for large softmax normalization over batch entries. CLIP relies on a symmetric contrastive loss that computes a full $N \times N$ similarity matrix across all image-text pairs in a batch, requiring both *image-to-text* and *text-to-image* losses. This results in quadratic computational complexity and necessitates extremely large batch sizes (32k+) to maintain a diverse set of negative samples.

In contrast, SigLIP replaces this formulation with a **pairwise sigmoid loss**, which allows each image-text pair to be evaluated independently:

$$\mathcal{L}_{\text{SigLIP}} = -\frac{1}{|B|^2} \sum_{i,j \in B} \log \sigma(z_{ij} \cdot (t \cdot x_i^\top y_j - b))$$

where $z_{ij} \in \{+1, -1\}$ indicates whether the pair is a true match, t is a learnable temperature parameter, and b is a learnable bias. The use of the sigmoid function σ in this context means the model no longer depends on global normalization over the entire batch. Most importantly, SigLIP performs robustly with much smaller batch sizes—enabling high-quality training with batch sizes between 2k and 8k—making it a more accessible and efficient choice.

The SigLIP vision encoder of Gemma 3 only processes images of size 896 × 896 pixels. If an image has a different aspect ratio, a "Pan and Scan" strategy is applied to crop it into one or more square patches. Each patch is passed through the SigLIP ViT encoder and then passed through a multimodal projector, essentially a Linear layer, before being interleaved into the text embeddings for causal generation.

During generation, Gemma 3 uses a single decoder-only transformer that attends over both visual and textual tokens. Visual tokens receive **bidirectional attention** allowing image patch tokens to see all other patches in the same image, while text tokens are attended to causally. Text tokens can freely attend to prior text and visual tokens but not future text, preserving autoregressive generation.

An overview of the Gemma 3 architecture, including the SigLIP-based vision encoder, multimodal token projector, and unified decoder, is illustrated in Figure 5.6.

Exercise 5.2 *As the final exercise, we will be implementing a function for multimodal tokenisation and testing multimodal generation with Gemma 3. This exercise will guide you through completing the corresponding notebook located at* `lxmls/labs/notebooks/multimodal/vlm_exercise.ipynb`*, focusing on the key steps required to process interleaved image and text data for a VLM.*

1. ***Setup:*** *Begin by running the first cell in the notebook to import the necessary libraries.*

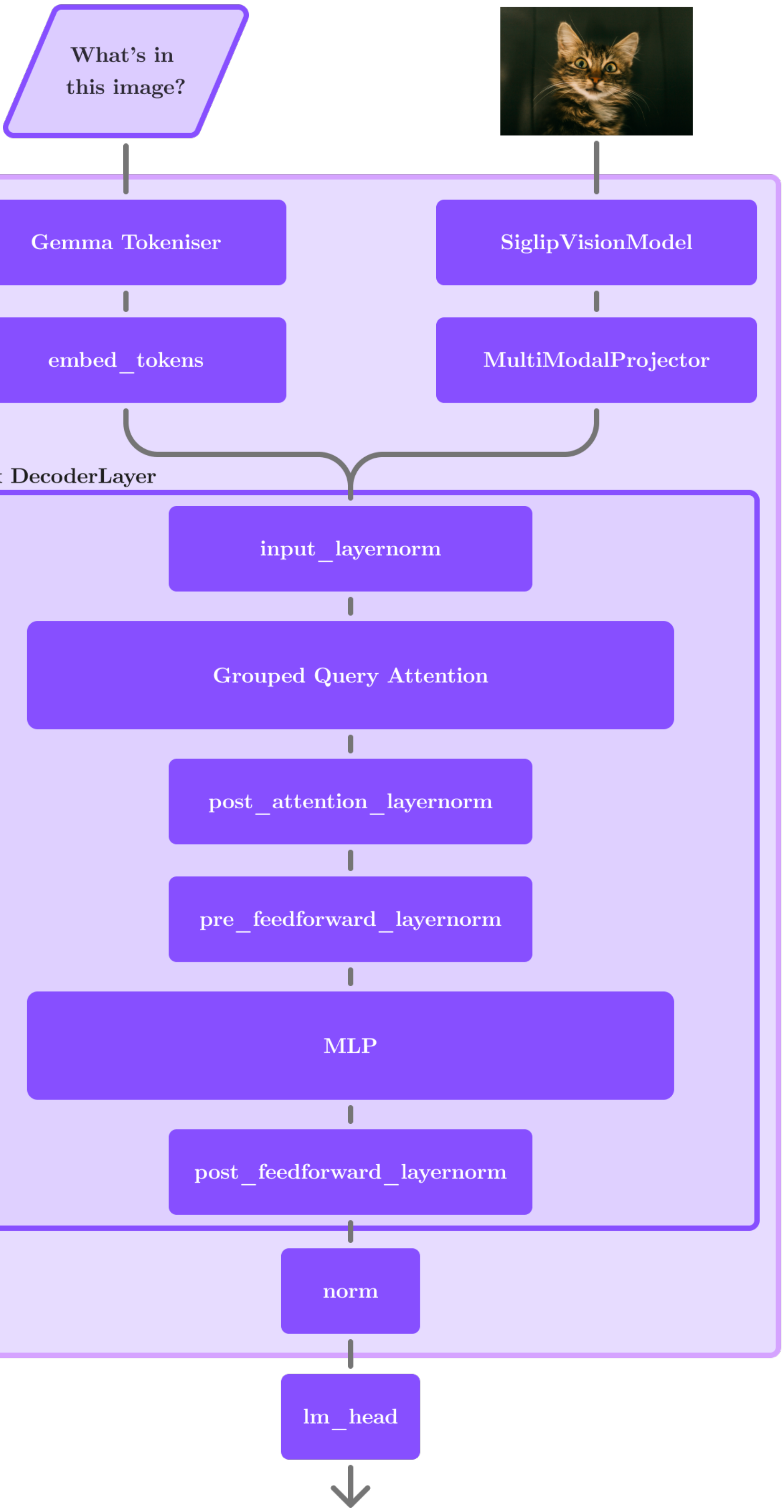


Figure 5.6: Architecture diagram of Gemma 3 (4B variant), highlighting the visual encoder, token projector, and decoder with mixed attention.

2. ***Tokenisation without padding using Image Placeholders***
   Your first task is to complete the `tokenize_without_padding` *function which translates a multimodal sequence of text and images into a list of token IDs. Your implementation should:*

   - *Start the token sequence with a single beginning-of-sequence (* `[BOS]` *) token.*

- *Iterate through the prompt. If an element is a string, encode it into token IDs. If it is an image, you must insert a specific sequence of special tokens: a newline, a beginning-of-image token, a set of image placeholder tokens, and an end-of-image token. Each of these tokens are accessed as part of the* `Tokenizer` *class defined in "lxmls/multimodal/gemma3/processor.py".*
- *As hinted in the notebook, the number of placeholder tokens is determined by the vision model's architecture. It corresponds to the number of feature vectors the SigLIP encoder generates for each image. Refer to the definition of the SigLIP encoder in "lxmls/multimodal/gemma3/siglip_vision/model.py" and its config in "lxmls/multimodal/gemma3/siglip_vision/model.py" to find this value.*
- *Finally, collect all PIL Images into a separate list to be processed by the vision encoder later.*

```
def tokenize_without_padding(
    prompt: Sequence[str | torch.Tensor],
    tokenizer: Tokenizer,
    vis_cfg: SiglipVisionModelConfig,
) -> Tuple[list[int], list[torch.Tensor]]:
    """
    Processes a single preprocessed prompt containing text and image tensors.
    """
    token_ids: List[int] = []
    images: List[torch.Tensor] = []

    # YOUR SOLUTION HERE

    return token_ids, images
```

3. ***Padding the Sequences***
   Most deep learning models process data in batches for efficiency. Since different prompts will have different lengths of text and numbers of images, we must pad them to uniform dimensions.

   - *Complete the* `pad_token_sequences` *function to pad each token list to the maximum sequence length in the batch using the tokenizer's* `pad_id`*.*
   - *Next, complete the* `pad_image_sequences` *function. This function should pad the list of images with zero-tensors and create a boolean* `image_presence_mask` *to differentiate real images from padding.*

```
def pad_token_sequences(all_token_ids: List[List[int]], max_seq_len: int, pad_id:
    int) -> List[List[int]]:
    """
    Pads each token sequence in a batch to a specified maximum length.
    """
    # YOUR SOLUTION HERE

    return finalised_token_ids

def pad_image_sequences(
    all_images: List[List[torch.Tensor]],
    max_num_images: int,
    vis_cfg: SiglipVisionModelConfig,
    device: Any,
) -> Tuple[List[List[torch.Tensor]], List[List[bool]]]:
    """
    Pads each list of images in a batch to a specified maximum number.
    """
    # YOUR SOLUTION HERE

    return image_batch, image_presence_mask
```

4. ***Execution and Generation:*** *The notebook provides the main* `tokenize` *and* `generate` *functions that orchestrate the steps you've implemented.*

- *Review the logic in these functions to understand how your helper functions are used in the complete pipeline.*
- *Run the final cells to test your implementation. Observe the model's responses to the text-only, single-image, and interleaved prompts.*
- *How does the model's output for the final prompt demonstrate its ability to ground language in visual context?*

Bibliography

Bahdanau, D., Cho, K., and Bengio, Y. (2014). Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*.

Bai, Y., Jones, A., Ndousse, K., Askell, A., Chen, A., DasSarma, N., Drain, D., Fort, S., Ganguli, D., Henighan, T., Joseph, N., Kadavath, S., Kernion, J., Conerly, T., El-Showk, S., Elhage, N., Hatfield-Dodds, Z., Hernandez, D., Hume, T., Johnston, S., Kravec, S., Lovitt, L., Nanda, N., Olsson, C., Amodei, D., Brown, T., Clark, J., McCandlish, S., Olah, C., Mann, B., and Kaplan, J. (2022). Training a helpful and harmless assistant with reinforcement learning from human feedback.

Bertsekas, D., Homer, M., Logan, D., and Patek, S. (1995). *Nonlinear programming*. Athena Scientific.

Bishop, C. (2006). *Pattern recognition and machine learning*, volume 4. Springer New York.

Blitzer, J., Dredze, M., and Pereira, F. (2007). Biographies, bollywood, boom-boxes and blenders: Domain adaptation for sentiment classification. In *Annual Meeting-Association For Computational Linguistics*, volume 45, page 440.

Boyd, S. and Vandenberghe, L. (2004). *Convex optimization*. Cambridge Univ Pr.

Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J. D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., et al. (2020a). Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901.

Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D. M., Wu, J., Winter, C., Hesse, C., Chen, M., Sigler, E., Litwin, M., Gray, S., Chess, B., Clark, J., Berner, C., McCandlish, S., Radford, A., Sutskever, I., and Amodei, D. (2020b). Language models are few-shot learners. *CoRR*, abs/2005.14165.

Chen, T., Kornblith, S., Norouzi, M., and Hinton, G. (2020). A Simple Framework for Contrastive Learning of Visual Representations. arXiv:2002.05709 [cs].

Cover, T., Thomas, J., Wiley, J., et al. (1991). *Elements of information theory*, volume 6. Wiley Online Library.

Crammer, K., Dekel, O., Keshet, J., Shalev-Shwartz, S., and Singer, Y. (2006). Online Passive-Aggressive Algorithms. *JMLR*, 7:551–585.

Crammer, K. and Singer, Y. (2002). On the algorithmic implementation of multiclass kernel-based vector machines. *The Journal of Machine Learning Research*, 2:265–292.

Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. (2018). Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*.

Dosovitskiy, A., Beyer, L., Kolesnikov, A., Weissenborn, D., Zhai, X., Unterthiner, T., Dehghani, M., Minderer, M., Heigold, G., Gelly, S., Uszkoreit, J., and Houlsby, N. (2021). An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale. arXiv:2010.11929 [cs] version: 2.

Duda, R., Hart, P., and Stork, D. (2001). *Pattern classification*, volume 2. Wiley New York.

Jaynes, E. (1982). On the rationale of maximum-entropy methods. *Proceedings of the IEEE*, 70(9):939–952.

Joachims, T. (2002). *Learning to Classify Text Using Support Vector Machines: Methods, Theory and Algorithms*. Kluwer Academic Publishers.

Kamath, G. A., Ferret, J., Pathak, S., Vieillard, N., Merhej, R., Perrin, S., Matejovicova, T., Ramé, A., Rivière, M., Rouillard, L., Mesnard, T., Cideron, G., Grill, J.-B., Ramos, S., Yvinec, E., Casbon, M., Pot, E., Penchev, I., Liu, G., Visin, F., Kenealy, K., Beyer, L., Zhai, X., Tsitsulin, A., Busa-Fekete, R., Feng, A., Sachdeva, N., Coleman, B., Gao, Y., Mustafa, B., Barr, I., Parisotto, E., Tian, D., Eyal, M., Cherry, C., Peter, J.-T., Sinopalnikov, D., Bhupatiraju, S., Agarwal, R., Kazemi, M., Malkin, D., Kumar, R., Vilar, D., Brusilovsky, I., Luo, J., Steiner, A., Friesen, A., Sharma, A., Sharma, A., Gilady, A. M., Goedeckemeyer, A., Saade, A., Kolesnikov, A., Bendebury, A., Abdagic, A., Vadi, A., György, A., Pinto, A. S., Das, A., Bapna, A., Miech, A., Yang, A., Paterson, A., Shenoy, A., Chakrabarti, A., Piot, B., Wu, B., Shahriari, B., Petrini, B., Chen, C., Lan, C. L., Choquette-Choo, C. A., Carey, C., Brick, C., Deutsch, D., Eisenbud, D., Cattle, D., Cheng, D., Paparas, D., Sreepathihalli, D. S., Reid, D., Tran, D., Zelle, D., Noland, E., Huizenga, E., Kharitonov, E., Liu, F., Amirkhanyan, G., Cameron, G., Hashemi, H., Klimczak-Plucińska, H., Singh, H., Mehta, H., Lehri, H. T., Hazimeh, H., Ballantyne, I., Szpektor, I., Nardini, I., Pouget-Abadie, J., Chan, J., Stanton, J., Wieting, J., Lai, J., Orbay, J., Fernandez, J., Newlan, J., yeong Ji, J., Singh, J., Black, K., Yu, K., Hui, K., Vodrahalli, K., Greff, K., Qiu, L., Valentine, M., Coelho, M., Ritter, M., Hoffman, M., Watson, M., Chaturvedi, M., Moynihan, M., Ma, M., Babar, N., Noy, N., Byrd, N., Roy, N., Momchev, N., Chauhan, N., Bunyan, O., Botarda, P., Caron, P., Rubenstein, P. K., Culliton, P., Schmid, P., Sessa, P. G., Xu, P., Stanczyk, P., Tafti, P., Shivanna, R., Wu, R., Pan, R., Rokni, R., Willoughby, R., Vallu, R., Mullins, R., Jerome, S., Smoot, S., Girgin, S., Iqbal, S., Reddy, S., Sheth, S., Põder, S., Bhatnagar, S., Panyam, S. R., Eiger, S., Zhang, S., Liu, T., Yacovone, T., Liechty, T., Kalra, U., Evci, U., Misra, V., Roseberry, V., Feinberg, V., Kolesnikov, V., Han, W., Kwon, W., Chen, X., Chow, Y., Zhu, Y., Wei, Z., Egyed, Z., Cotruta, V., Giang, M., Kirk, P., Rao, A., Lo, J., Moreira, E., Martins, L. G., Sanseviero, O., Gonzalez, L., Gleicher, Z., Warkentin, T., Mirrokni, V., Senter, E., Collins, E., Barral, J., Ghahramani, Z., Hadsell, R., Matias, Y., Sculley, D., Petrov, S., Fiedel, N., Shazeer, N., Vinyals, O., Dean, J., Hassabis, D., Kavukcuoglu, K., Farabet, C., Buchatskaya, E., Alayrac, J.-B., Anil, R., Lepikhin, D., Borgeaud, S., Bachem, O., Joulin, A., Andreev, A., Hardin, C., Dadashi, R., and Hussenot, L. (2025). Gemma 3 technical report. *arXiv preprint arXiv:2503.19786*.

Lewis, M., Liu, Y., Goyal, N., Ghazvininejad, M., Mohamed, A., Levy, O., Stoyanov, V., and Zettlemoyer, L. (2020). BART: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 7871–7880, Online. Association for Computational Linguistics.

Liu, Y., Ott, M., Goyal, N., Du, J., Joshi, M., Chen, D., Levy, O., Lewis, M., Zettlemoyer, L., and Stoyanov, V. (2019). Roberta: A robustly optimized bert pretraining approach.

Manning, C., Raghavan, P., and Schütze, H. (2008). *Introduction to information retrieval*, volume 1. Cambridge University Press Cambridge, UK.

Manning, C. and Schütze, H. (1999). *Foundations of statistical natural language processing*, volume 59. MIT Press.

Mitchell, T. (1997). *Machine learning*.

Nocedal, J. and Wright, S. (1999). *Numerical optimization*. Springer verlag.

OpenAI (2023). Gpt-4 technical report.

Ouyang, L., Wu, J., Jiang, X., Almeida, D., Wainwright, C. L., Mishkin, P., Zhang, C., Agarwal, S., Slama, K., Ray, A., Schulman, J., Hilton, J., Kelton, F., Miller, L., Simens, M., Askell, A., Welinder, P., Christiano, P., Leike, J., and Lowe, R. (2022). Training language models to follow instructions with human feedback.

Peters, M. E., Neumann, M., Iyyer, M., Gardner, M., Clark, C., Lee, K., and Zettlemoyer, L. (2018). Deep contextualized word representations.

Radford, A., Kim, J. W., Hallacy, C., Ramesh, A., Goh, G., Agarwal, S., Sastry, G., Askell, A., Mishkin, P., Clark, J., Krueger, G., and Sutskever, I. (2021). Learning Transferable Visual Models From Natural Language Supervision. arXiv:2103.00020 [cs].

Radford, A., Narasimhan, K., Salimans, T., and Sutskever, I. (2018). Improving language understanding by generative pre-training.

Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., Sutskever, I., et al. (2019). Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9.

Raffel, C., Shazeer, N., Roberts, A., Lee, K., Narang, S., Matena, M., Zhou, Y., Li, W., and Liu, P. J. (2020). Exploring the limits of transfer learning with a unified text-to-text transformer.

Rosenblatt, F. (1958). The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386.

Schölkopf, B. and Smola, A. J. (2002). *Learning with Kernels*. The MIT Press, Cambridge, MA.

Sennrich, R., Haddow, B., and Birch, A. (2016). Neural machine translation of rare words with subword units. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1715–1725, Berlin, Germany. Association for Computational Linguistics.

Shalev-Shwartz, S., Singer, Y., and Srebro, N. (2007). Pegasos: Primal estimated sub-gradient solver for svm. In *ICML*.

Shannon, C. (1948). A mathematical theory of communication. *Bell Syst. Tech. Journ.*, 27(379):623.

Shawe-Taylor, J. and Cristianini, N. (2004). *Kernel Methods for Pattern Analysis*. CUP.

Vapnik, N. V. (1995). *The Nature of Statistical Learning Theory*. Springer-Verlag, New York.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I. (2017). Attention is all you need. *Advances in neural information processing systems*, 30.

Zhai, X., Mustafa, B., Kolesnikov, A., and Beyer, L. (2023). Siglip: Scaling language-image pretraining with a simple and effective loss. *arXiv preprint arXiv:2303.15343*.