# MF810: Environment setup and fundamentals

# Desirable properties of a program or code

Some important features of a program and its source code:

| | |
|---|---|
| **Efficiency** | The most obvious: Do what we want without taking too long. |
| **Correctness** | Do what we want in all circumstances, including edge cases. |
| | Note that in unexpected cases it is often better to abort than to just continue. |
| **Robustness** | Handle edge cases, unexpected inputs, or errors gracefully. |
| **Maintainability** | Adding features, changing features, or fixing bugs should be easy. |
| **Readability** | Somebody else (or future you) should be able to understand the code. |
| | Not only about **what** the code does, but also **why**. |
| | Helps maintainability. |
| **Portability** | The program should run correctly on other computers than the developer's. |

# Today

- Docker and Linux basics
- Python Docker setup
- Intro to Pytorch
- Intro to neural networks
- Neural neural networks in Pytorch

## Portability: Docker and Linux

- When moving code from one computer to another, it can fail to run. Some reasons include
  - Dependencies are not installed. package Xexist on another
  - Package versions differ.
  - Files are not in the right location.
  - Etc.
- These issues are solved by **containers**. (They also provide some safety/security benefits if used correctly.)

solve problem

- **Docker** is a system for running containers.
- Docker is built on features of the Linux kernel. To be proficient Docker users, we must build basic understanding of Linux topics.

# What is Linux?

Linux

- **Linux** typically refers to an operative system, like Windows or macOS.
- Strictly speaking, Linux is the **Linux kernel**.
- A **kernel** is the piece of software that coordinates everything in the background and communicates with the hardware. As far as software is concerned, the kernel is the computer!
- Examples include: the *Windows NT Kernel* on Windows; *XNU* on Apple devices; the *Linux kernel* on Android, etc.
- The Linux kernel provides features to run kernels inside the kernel itself,[1] which is what makes Docker and containerization possible.

[1] This is technically not what happens, but it is *similar* to doing so.

# Foundations

Let us look at basics of how a Linux environment works.

Windows and macOS are in many ways very similar, but details differ.

We will discuss

- Users, groups, and permissions.
- Filesystems and file permissions.
- Environment variables.
- The “command line.”

## Users and groups

- When setting up a new computer, you must enter a (user)name.
  In this process, the OS sets up a **user** (account) for you.
- Systems typically have one **super user**[2] and one or more regular users.
- The purpose of multiple users is to restrict users from doing certain things:
  - One users should not be able to read other users' files.
  - Only the super user (admin) should be granted unlimited access to the computer.
  - Etc.
- *Different users have different* $\boxed{\textbf{permissions}}$.
- Users can be members of $\boxed{\textbf{groups}}$, which may include multiple users.
  Groups are another tool to grant permissions. For instance, users who are members of the printer group are allowed to use the printers.

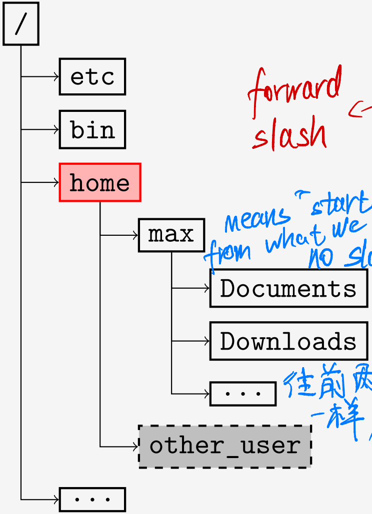
superuser

Groups1 — user1, user2 → diff

Groups2 — user → permission.

[2] **root** user on Linux/Unix/macOS or *Administrator* on Windows.

# Filesystems




forward slash

means "start from what we are" no slash

往前两个一样

- The **filesystem** is a sequence of **directories** (folders) starting at the **root** (`/` on Linux/Unix/macOS; like `C:/` on Windows).
- A file location is given by a **path**:
  `/home/max/Documents/MF810/lecture1.pdf`
  This is called an **absolute path** because it references the file from the root.
- A **relative path** is relative to the current directory. From `/home/max` this relative path is the same file:
  `Documents/MF810/lecture1.pdf`
- Special path components: `./` means same/current directory; 即为当前位置. `../` means **parent directory** ("up/back").
- From `/home/max/Downloads`, these are still the same files:
  /home/max
  `../Documents/MF810/lecture1.pdf` and
  `./../Documents/MF810/./lecture1.pdf`
- Relative paths should be used within a project!

# Files and file permissions

- A directory may contain a number of **files**.
- There are three main types of files: regular files, symlinks (references/shortcuts), and directories.
- Each file has an **owner**, a **group owner**, and two sets of permissions: owner permissions and group permissions.
- A file listing may look like

$$\underbrace{\texttt{-}}_{\text{type (-,l,d)}}\underbrace{\texttt{rw-r--r--}}_{\text{permissions}}\ \texttt{1}\ \underbrace{\texttt{max}}_{\text{owner}}\ \underbrace{\texttt{users}}_{\text{group}}\ \underbrace{\texttt{206K}}_{\text{size}}\ \underbrace{\texttt{Jan 13 14:23}}_{\text{date}}\ \underbrace{\texttt{lecture1.pdf}}_{\text{filename}}$$

owner

run (x)

`rw-` indicates that the owner (max) may read (`r`) and write (`w`) to the file.

other

`r--` indicates that all members of the group users may read (`r`) but not write (`w`) to the file.

- Permissions are central to the system and a great protection against accidental mistakes.

## Docker images

- Docker **images** and **containers** are like classes and objects in object-oriented programming: containers are specific instances of images.
- Images can be created in two ways:
  - By downloading an existing image, e.g. `docker pull ubuntu` or `docker pull python:slim`.
  - By extending an existing image in a `Dockerfile` and building it: `docker build -t image_name path_to_Dockerfile_dir`
    We will use the name `my_python_im` and the path will be `.` if run from the same directory as the `Dockerfile`.
- Containers are created from an image using either `docker run` or `docker create`.
- Making changes to a container does not change the image it was created from!

# Containers: `docker run` example

- Once set up, we run any Python script by entering

  ```
  docker run --rm -it my_python_im -v "$PWD":/workdir python scriptname.py
  ```

  - `docker run` creates and runs a container from an **image**.
  - `--rm` deletes the container after termination.
  - `-it` specifies interactive mode (as opposed to running in the background).
  - my `python_image` is the name we have given our Python image.
  - `-v "$PWD":/workdir` makes the current directory (`$PWD`) available in the container at the `/workdir` path.
  - `python` specifies that the container should execute the Python executable.
  - `scriptname.py` is the script file we want python to run.
- This is long, but we can make a shortcut as e.g.

  ```
  dockpy scriptname.py
  ```

- Like this, your code can be sent to a friend or a **cloud computer** and run exactly like it does on your computer!

# Deep learning libraries and neural networks

# Differentiation and optimization

- What are famous libraries like Tensorflow and Pytorch?
- Short answer:

  They are numerical libraries with extra support for differentiating special functions.
- Consider an abstract problem of minimizing a function $L$ as a function of $\theta$:

$$\min_{\theta \in \mathbb{R}^P} L(\theta).$$

- If $L$ is smooth, the first order condition at the minimum states

* --> 表示optimal.

$$\nabla_\theta L(\theta)\Big|_{\theta=\theta^*} = \nabla_\theta L(\theta^*) = 0.$$

- $\boxed{\textit{Gradients}}$ *are intimately connected to optimization, and that is the interest in numerical libraries with differentiation support.*

## Gradient descent

- Recall from calculus, that the gradient $\boxed{\nabla_\theta L}$ is a $\boxed{\text{vector}}$ pointing in the direction in which the function $L$ is most increasing (steepest).
- Similarly, $-\nabla_\theta L = \nabla_\theta(-L)$ is the direction in which $-L$ is most increasing, i.e., $-\nabla_\theta L$ is the direction that $L$ is most decreasing.
- If we from any point $\theta$ move a small distance $\epsilon \nabla_\theta L(\theta)$ along the gradient to $\theta'$, we should have $\boxed{L(\theta') \leq L(\theta)}$.
- We have discovered **gradient descent**:

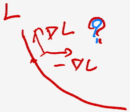
at point minus move a small distance.

$$\left[ \quad \theta_{n+1} = \theta_n - \epsilon \nabla_\theta L(\theta_n). \quad \right]$$

$\epsilon$ : → learning rate

$\epsilon$

- Because our deep learning library can compute gradients, this is trivial to implement!
- The parameter $\epsilon$ is called the **learning rate**.




# Our first example: linear regression

- Consider the problem of least squares linear regression:

x is vector

$x_i$, $i \in \{1, \dots, N\}$

$\in \mathbb{R}^P$

$$\min_{\theta \in \mathbb{R}^P} \sum_{i=1}^{N} \ell(y_i, f(x_i; \theta))$$

where

$|y_i - \theta \cdot x|^2$ inner product

$$\ell(y, y') = |y - y'|^2 \quad \text{and} \quad f(x; \theta) = \theta \cdot x. = \sum_{j=1}^{P} \theta^j x^j$$

- This is called linear regression because $f$ is linear in $\theta$, not because it is linear in $x$!
- Define

$$L(\theta) = \sum_{i=1}^{N} \ell(y_i, f(x_i; \theta))$$

same as $\min L(\theta)$

so that ↙ so take gradient

$$\nabla_\theta L(\theta) = \sum_{i=1}^{N} \partial_{y'} \ell(y_i, f(x_i; \theta)) \nabla_\theta f(x_i; \theta) = \sum_{i=1}^{N} 2(y_i - x_i \cdot \theta) x_i.$$

- These expressions are much simpler on matrix form.

# Linear regression gradient descent

- With

$$\nabla_\theta L(\theta) = \sum_{i=1}^{N} 2(y_i - x_i \cdot \theta)x_i$$

gradient descent is equivalent to iterating

start with

$$\theta_{n+1} = \theta_n - \epsilon \underbrace{\sum_{i=1}^{N} 2(y_i - x_i \cdot \theta)x_i}.$$

times with

- For linear regression, this method converges to the minimizer $\theta^*$.[3]

[3] If $\epsilon$ is decreasing at an appropriate rate.

# Linear/dense layers

- The function $f(x;\theta) = \theta \cdot x = \theta x$ maps $x \in \mathbb{R}^P$ to $\mathbb{R}$.
- Generalize this to $\mathbb{R}^P$ to $\mathbb{R}^k$ by taking $\boxed{\theta \in \mathbb{R}^{P\times k}}$:

$$\mathbb{R}^P \to \mathbb{R}^k \qquad x \mapsto \theta x.$$

- It is useful to apply another function $\sigma$ to the output.
- Often $\sigma : \mathbb{R} \to \mathbb{R}$, in which case this is done element-by-element, and we write $\sigma.$ for such **broadcasting**:

$$\sigma.(x) = (\sigma(x_1), \ldots, \sigma(x_k)), \quad x \in \mathbb{R}^k.$$

  We write just $\sigma(x)$ to mean $\sigma.(x)$ in this case.
- The transformation $x \mapsto \theta x$ is called a **linear layer**.
- The function $f(x;\theta) = \sigma(\theta x)$ is called just a **layer** or a **dense layer**.

# Neural networks

- The function $\boxed{x \mapsto \sigma(\theta x)}$ is an example of a trivial (**artificial**) **neural network** (**ANN** or **NN**).
- Let $\theta = (w_1, w_2)$ and define $h_1(x; w_1) = \sigma_1(w_1 x)$ and $h_2(x; w_2) = \sigma_2(w_2 x)$. The composition (assuming $w_1 \in \mathbb{R}^{k_0 \times k_1}$ and $w_2 \in \mathbb{R}^{k_1 \times k_2}$)

$$(h_2 \circ h_1)(x) = h_2(h_1(x)) = \sigma_2(w_2 \sigma_1(w_1 x))$$

  is a **two-layer neural network**.
- The function $h_1$ is called a **hidden layer** and the function $h_2$ is called the **output layer**.
- The parameters $w_1$ and $w_2$ are called **weights**.
- The functions $\sigma_1$ and $\sigma_2$ are called **activation functions**.

# Neuron bias

- For reasons we discuss later, it is common to work with affine transformations instead of linear.
- Let $\theta = ((w_1, b_1), \ldots, (w_d, b_d))$ and define

each adding b.

$$h_i(x) = \sigma_i(b_i + w_i x).$$

- The term $b_i$ is called a **neuron bias**, often **bias** for short.
- The composition $h_d \circ h_{d-1} \circ \cdots \circ h_2 \circ h_1$ is a **neural network** of **depth** $d$.
- Because $h_d(x)$ is often a linear transformation to a scalar ($\mathbb{R}^{k_{d-1}} \to \mathbb{R}^{k_d} = \mathbb{R}$), it is customary to speak of the number of **hidden layers**, which here is $d - 1$.
- Like before, the weights must satisfy $w_i \in \mathbb{R}^{k_{i-1} \times k_i}$ and $b \in \mathbb{R}^{k_i}$. The value $k_i$ is called the **width** of layer $i$.
- The number of parameters in this network is

depth * width ?

$$P = \sum_{i=1}^{d} (k_{i-1} + 1) k_i.$$

# MLPs and more

- This type of neural network is called a **multilinear perceptron** (**MLP**).
- An MLP is a **feedforward** neural network in which all neurons in two adjacent layers are connected to each other (called **fully connected** or **dense**).
- There are many other examples of layers. Some examples include

  **Convolutional layers** Feedforward but not fully connected.
  Commonly used to find spatial patterns, like in images.

  **Recurrent layers** Not feedforward.
  Used for sequential data like natural language processing (NLP) and time series.

  **Dropout layers** Randomly omits connections: feedforward but not dense.
  Used for regularization.
- Such common structures are included in the libraries and can be used directly.
- There are no restrictions on general layers; custom layers are often created.