

# **Sviluppo e ciclo vitale di software di intelligenza artificiale**

Angelo Vaccaro

# Contents

<b>1 Pre Requisiti per il corso</b>	<b>3</b>
<b>2 Obiettivi del corso</b>	<b>3</b>
<b>3 Programma del corso</b>	<b>3</b>
<b>4 Valutazione del corso</b>	<b>4</b>
<b>5 Version Control Systems (VCS)</b>	<b>4</b>
5.1 Source Control . . . . .	4
5.2 Git . . . . .	5
5.3 Commit . . . . .	5
5.4 Git CLI vs IDE . . . . .	7
5.5 Install requirements . . . . .	7
5.6 Visual Studio Code + Git . . . . .	7
5.7 VSC + Git - Collaboration . . . . .	14
5.8 Branches . . . . .	15
5.9 Merging . . . . .	15
<b>6 DevOps vs MLOps</b>	<b>15</b>
6.1 Passaggio dal DevOps al MLOps . . . . .	16
6.1.1 Continuous Integration . . . . .	17
6.1.2 Continuous Delivery . . . . .	18
6.2 MLOps . . . . .	18
6.3 Esempio (e corso accelerato di Python) . . . . .	20
6.3.1 MakeFile . . . . .	20
6.3.2 Requirements.txt . . . . .	23
6.3.3 Test script . . . . .	23
6.3.4 Continous Integration . . . . .	25
<b>7 Data Science and EDA</b>	<b>26</b>
7.1 Data types: quantitative . . . . .	27
7.2 Exploratory Data Analysis (EDA) . . . . .	28
7.3 Preprocessing in DL . . . . .	31
7.4 Model Training & Selection . . . . .	31
7.4.1 Training . . . . .	37
7.5 Come includere PyTorch and EDA nel nostro progetto con DevOps .	40
7.6 Pytorch Lightning . . . . .	48
7.7 Come gestiamo i modelli che già esistono? . . . . .	50
<b>8 iperparametri</b>	<b>50</b>
8.1 Grid Search: Idea . . . . .	51

<b>9 Docker, Logging e Monitoring</b>	<b>53</b>
9.1 Docker . . . . .	53
9.2 Containerization . . . . .	54
9.2.1 Docker vs Virtual Machine . . . . .	55
9.2.2 Docker commands . . . . .	56
9.3 Immagine . . . . .	56
9.3.1 Docker Run . . . . .	57
9.3.2 DOckerfile . . . . .	58
9.4 Volumi . . . . .	61
9.5 Integrazione con VSCode del docker . . . . .	64
9.6 Allenare un modello in Docker . . . . .	65
9.7 Inferenza . . . . .	69
9.8 CI/CD con Docker . . . . .	70
<b>10 Logging e monitoring</b>	<b>72</b>
10.1 Log aggregation and training . . . . .	73
<b>11 Python Packaging</b>	<b>75</b>
<b>12 AutoML</b>	<b>84</b>
12.1 Caratteristiche principali di AutoML . . . . .	84
12.2 Vantaggi e limiti . . . . .	84
12.3 Esempi di strumenti AutoML . . . . .	85
12.4 Esempio pratico con Auto-sklearn . . . . .	85
<b>13 Tool utili</b>	<b>85</b>
<b>14 Progetto d'esame</b>	<b>87</b>

# 1 Pre Requisiti per il corso

- Conoscenza della bash del codice Unix/Linux
- Conoscenza di base di Python
  - operatori del linguaggio di programmazione (if/else, for loop, while, etc)
  - differenza tra compilare ed eseguire uno script
- Basi di Machine Learning e Deep Learning

# 2 Obiettivi del corso

Il corso prevede quello di affrontare un problema con Machine learning e deep learning. Cosa devo fare? Devo capire come muovermi e cos'è il mio problema analizzando i dati a disposizione. Una volta che ho risolto il mio problema devo farlo andare in sviluppo, per farlo funzionare su più macchine. Dobbiamo quindi chiederci come facciamo a sviluppare in modo sensato il software? Dopo averlo sviluppato, come facciamo a farlo funzionare in produzione?

# 3 Programma del corso

- Day 1
  - Version Control System (VCS)
    - \* Git
    - \* GitHub
  - Differenza tra DEVOps ed MLOps che sono i 2 processi di produzione del codice, uno nell'ambito di produzione normale mentre l'altro nell'ambito di machine learning e deep learning.
- Day 2
  - Data science, necessaria per investigare il problema e capire come risolverlo.
  - Capiremo come addestrare un modello from scratch, ovvero da zero, senza utilizzare librerie preconfezionate.
  - Piccolo accenno sul discorso degli iperparametri, che sono i parametri che dobbiamo settare per far funzionare il nostro modello.
- Day 3
  - Parleremo di Docker, che è un software che ci permette di creare dei container, ovvero delle macchine virtuali leggere che ci permettono di eseguire il nostro codice in modo isolato.

- Discorso di Logging e Monitoring, ovvero come tenere traccia di quello che succede nel nostro codice e come monitorare le performance del nostro modello.
- Day 4
  - Vedremo come si fa un packaging python, ovvero come creare un pacchetto python che possiamo distribuire e installare su altre macchine.
  - Un accenno al discorso di AutoML, ovvero come automatizzare il processo di addestramento del modello.

## 4 Valutazione del corso

- Bisognerà svolgere un progetto che affronti tutte le problematiche viste a lezione.
  - Come si fa DEVOps, quindi utilizzo di Git, lint, unit test, etc.
  - Training e test di un piccolo modello di Machine Learning.
  - Si può implementare una piccola interfaccia di testing.
  - Dockerizzazione del progetto (importante).
  - GitHub Actions per il deploy del progetto.
  - Monitoring e Logging del progetto.

## 5 Version Control Systems (VCS)

Se dobbiamo sviluppare un software, dobbiamo tenere traccia delle modifiche che facciamo al codice. Per farlo utilizziamo i Version Control Systems (VCS), che ci permettono di tenere traccia delle modifiche, di collaborare con altri sviluppatori e di gestire le versioni del nostro codice.

### 5.1 Source Control

Cosa vuol dire Source Control? Vuol dire andare a sviluppare un progetto in modo che tutta questa struttura (progetti e file) sia organizzata. Se faccio una modifica e voglio tenerne traccia potrei essere tentato di fare copia e incolla, portandoci ad un aumento esponenziale del numero di file e della loro gestione. I VCS sono in grado di tenere traccia delle modifiche che facciamo il codice andando a fare:

- **Restore** → ripristinare una versione precedente delle modifiche che stiamo facendo.
- **Check Out** → andare a vedere le modifiche che sono state fatte in un determinato file.
- **Recover** → recuperare un file che è stato cancellato.

- **Collaborate** → collaborare con altri sviluppatori, andando a gestire le modifiche che fanno loro e le modifiche che facciamo noi.

## 5.2 Git

Git è uno solo di questi VSC, ce ne sono molti altri ma utilizzeremo questo per il nostro progetto perchè è lo standard aziendale. Per parlare di Git dobbiamo specificare e chiarire bene il concetto di Repository. Che cos'è una Repository?

- una repository è l'archivio con tutti i cambiamenti che facciamo al nostro progetto.
- quando partiamo da una cartella vuota dobbiamo fare **git init** che dice a Git che questa cartella è una repository.
- fatto ciò verrà creata una cartella chiamata `.git`, in cui ci sono tutti i file che gestiscono il version del sistema, i branch e via dicendo.

## 5.3 Commit

Il commit è l'azione che ci permette di salvare le modifiche che abbiamo fatto al nostro progetto. Quando facciamo un commit, Git salva lo stato attuale del nostro progetto e lo aggiunge alla cronologia delle modifiche. Idealmente dovrebbe essere un singolo cambiamento, nella pratica delle cose i cambiamenti simili vengono accorpati. Tendenzialmente i cambiamenti si fanno quando siamo ad un buono stato del progetto. Il commit quindi lo possiamo vedere come un check point, non deve essere per forza funzionante ma è buona prassi che lo sia. Per far sì che il commit sia valido deve avere:

- **Autore:** chi ha fatto il commit, il nome e l'email.
- **Data:** (lo fa automaticamente Git)
- **Messaggio/Comment:** una breve descrizione del cambiamento che abbiamo fatto, che deve essere chiara e concisa.
- **Hashes:** un identificativo unico del commit, che viene generato automaticamente da Git. Questo hash è un numero esadecimale che identifica in modo univoco il commit.

La cosa più difficile da capire è il discorso della gestione delle versioni, ad esempio: Abbiamo 3 versioni e ne modifco 2 (file A e file C) ma non il file B. Quindi poi quando faccio la versione successiva avrò la versione modificata dei miei file ma anche la versione originale di B. Git gestisce questo discorso facendo la copia e non il delta. Il commit crea una catena di commit, che sono collegati tra loro. Ogni commit ha un hash che lo identifica in modo univoco e che è collegato al commit precedente. Come facciamo nell'effettivo questo commit? Lo si fa tramite una **staging area** :

- Prima di fare il commit dobbiamo dire a Git quali file vogliamo includere nel commit.
- La staging area è un piccolo container virtuale in cui buttiamo dentro cose finchè non abbiamo finito di lavorare, facendo un nuovo commit.
- **Nota:** creare un **nuovo file** è un cambiamento al codice. Ogni volta che creiamo un nuovo file dobbiamo aggiungerlo manualmente, altrimenti lui non sa che esiste.

Se non voglio aggiungere qualcosa ho varie opzioni:

- Non lo metto nella staging area, ma è un metodo poco pratico, che rischia di portarci all'errore.
- **gitignore** → è un file (.gitignore) che contiene all'interno tutti i nomi dei file e le cartelle che vogliamo ignorare. Tendenzialmente si utilizza per:
  - non avere all'interno file di compilazione (poco utili nel source control).
  - file troppo grandi (5-10 MB), non vogliamo appesantire la Repository.
  - checkpoint dei modelli, datasets, zips, etc.

Esempio di file **.gitignore**:

```
#Ignore data files
#questo ignora tutta la cartella,
#non comparira' mai sul source control una cartella
#chiamata data.
data/

#!data/.keep    # -----this won't work,
#because we told git to ignore THE data folder
#questo serve perche' a volte noi non vogliamo caricare
#la cartella,
#ma abbiamo bisogno della cartella come placeholder.

#Ignore all files INSIDE checkpoints
checkpoints/**

#the following line will work, because we are
#telling git to ignore
#the checkpoints folder, but NOT the .keep file inside it
!checkpoints/.keep

#Ignore all .pth files
#pth e' uno dei framework di deep learning piu' usati,
#e' un formato di file che contiene i pesi del modello e
#le informazioni necessarie per ricreare il modello.
*.pth
```

```

#Ignore all .pyc files
#pyc sono file intermedi di compilazione di python,
#ovvero file che python crea quando eseguiamo uno script.
*.pyc

#Ignore pycache files
#pycache e' una cartella che contiene i file .pyc
--pycache_--

```

## 5.4 Git CLI vs IDE

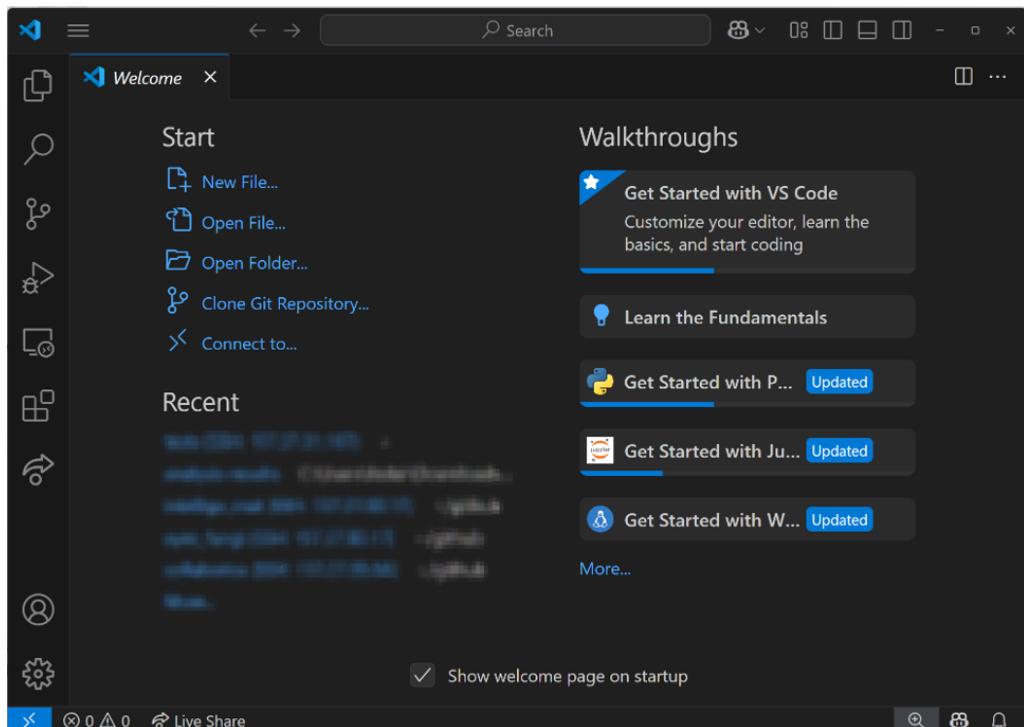
Git può essere utilizzato sia come command line ma si può utilizzare anche un IDE, noi utilizzeremo Visual Studio Code (non è propriamente un'IDE bensì un editor di testo). L'interfaccia di Visual Studio Code (come per qualsiasi IDE) può essere molto utile per l'utente.

## 5.5 Install requirements

- **Python:** si può utilizzare la versione di default di python preinstallata nel sistema, lo stesso vale per Ubuntu
- **Miniconda:** è una versione di python che fa anche la gestione degli ambienti virtuali, che sono delle istanze di python che possiamo utilizzare per isolare i nostri progetti. Ci permette di avere differenti interpreti di diverse versioni di python.
- **Visual Studio Code:** come detto prima è un editor di testo che possiamo utilizzare per scrivere il nostro codice. Ha un sacco di estensioni che ci permettono di utilizzare Git, Python, etc.
- **Git:** è il software che ci permette di utilizzare Git, ovvero il VCS che abbiamo visto prima. Possiamo installarlo tramite il package manager del nostro sistema operativo (apt per Ubuntu, brew per MacOS, etc).

## 5.6 Visual Studio Code + Git

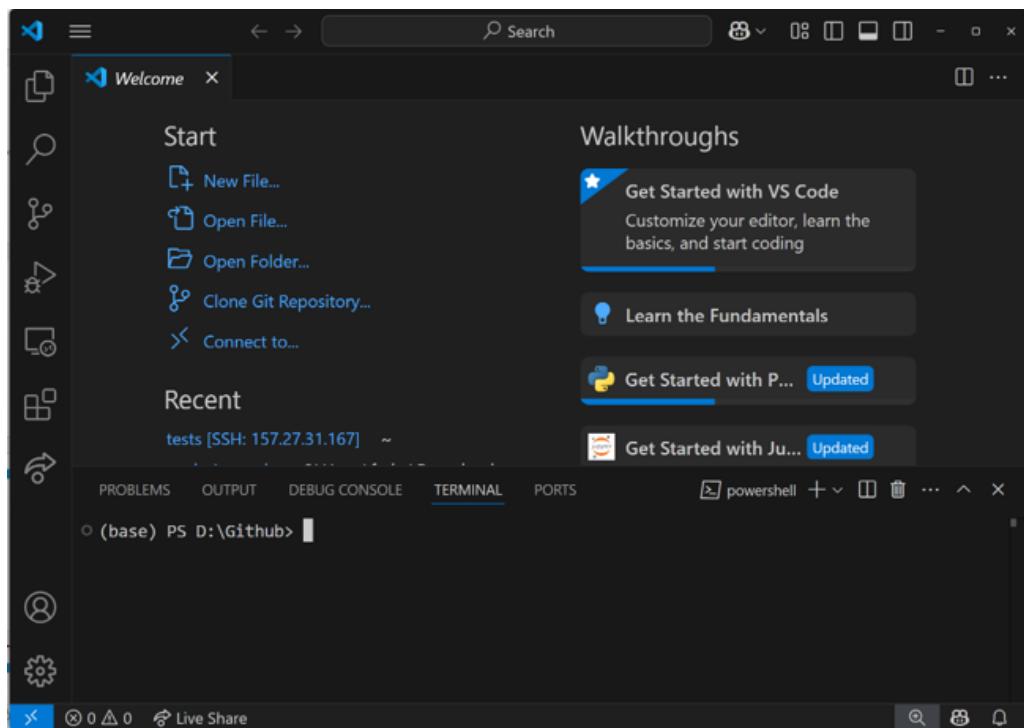
Visual Studio Code ha un'integrazione con Git che ci permette di utilizzare Git direttamente dall'editor. Possiamo fare i commit, le modifiche, le pull request, etc. direttamente dall'editor senza dover utilizzare la command line.



↑

Questa è la "Home Page di Visual Studio Code", dove possiamo vedere i file che abbiamo aperto, le modifiche che abbiamo fatto, i commit che abbiamo fatto, etc. Da qui tramite i pulsanti cmd+j possiamo aprire il terminale interno di Visual Studio Code.

↓



Ora nel terminale digitiamo il comando `mkdir git_tests` per creare una cartella di prova chiamata appunto `git_tests`. Per aprire la cartella in questione in alto a destra dovremo selezionare **Open Folder** e selezionare la cartella che abbiamo appena creato. Ora inizializziamo la repository vuota con il comando sul terminale `git init`. Noi questa cartella non la vediamo da VSC, nell'explorer possiamo vederla e troveremo dentro dei log, dei riferimenti ai commit e molto altro. Per esempio c'è il file config dove troviamo le informazioni del nostro utente, il file HEAD che ci dice a quale branch siamo collegati, etc. Iniziamo creando un primo file python:::

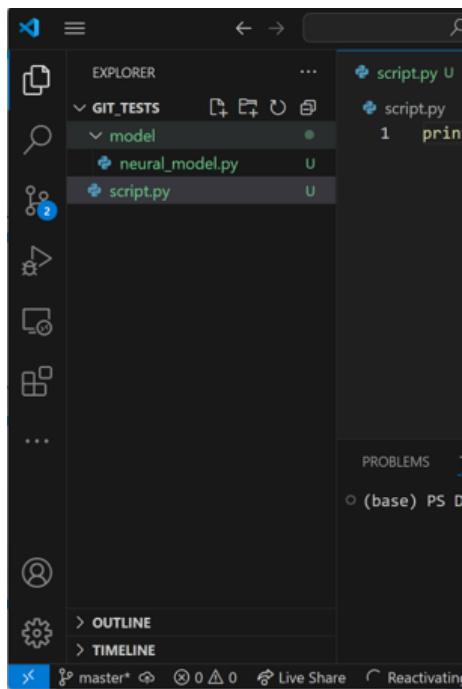


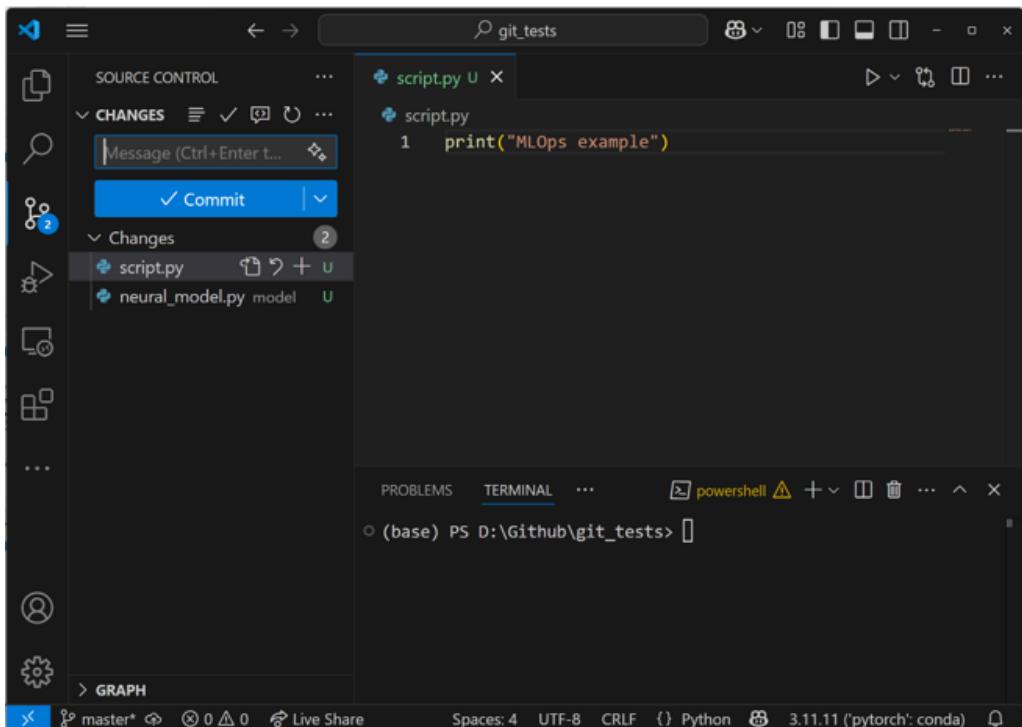
Notiamo subito delle modifiche nella nostra schermata di VSC. Questi simboli hanno un significato diverso in base a come Git sta trattando il nostro file:

- **A** → il file è stato aggiunto alla staging area, quindi è pronto per essere committato.
- **M** → il file è stato modificato, quindi è stato cambiato rispetto all'ultima versione.
- **D** → il file è stato cancellato, quindi non esiste più nella repository.
- **U** → il file è in uno stato di uncommitted changes, ovvero non è stato ancora aggiunto alla staging area.
- **C** → il file è in uno stato di conflicted changes, ovvero ci sono dei conflitti tra le modifiche che abbiamo fatto e quelle che sono state fatte da altri sviluppatori.

- **R** → il file è stato rinominato, quindi il nome del file è stato cambiato rispetto all'ultima versione.
- **S** → il file è stato spostato, quindi il file è stato spostato in un'altra cartella rispetto all'ultima versione.
- **T** → il file è cambiato da symlink a file regolare, o viceversa.

Aprendo sulla sinistra il source control possiamo vedere la parte di gestione di Git:





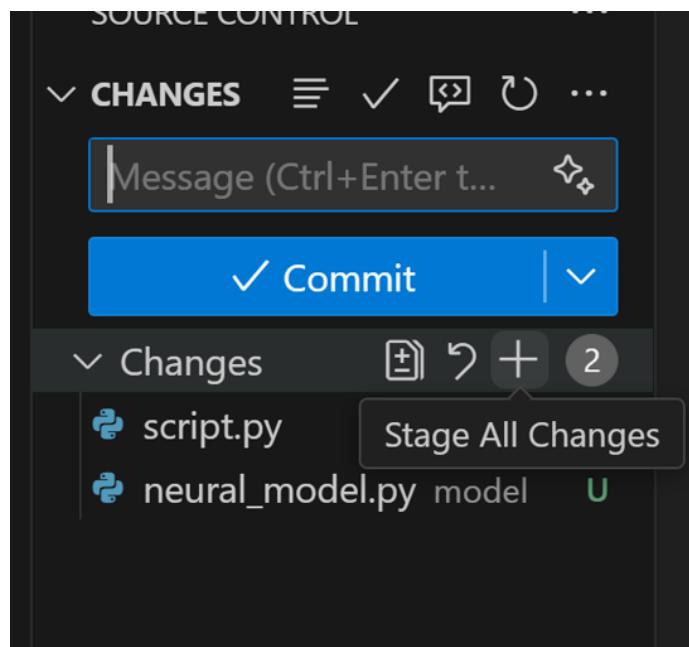
Qui vediamo che c'è la sezione **Changes** che ci mostra i cambiamenti che stiamo facendo, ed il pulsante **Commit** che serve appunto per creare un commit. Altrettanto importante è il pulsante + che ci permette di mettere in stage. Altro comando importante è ⌂ che ci permette di cancellare e fare undo dei cambiamenti, in questo caso con il file non tracciato (U), il file viene cancellato. Prima di fare qualcosa dobbiamo configurare il nostro utente, per farlo dobbiamo aprire il terminale e digitare i seguenti comandi:

↓

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL ...
● (base) PS D:\Github\git_tests> git config user.name authorname
● (base) PS D:\Github\git_tests> git config user.email mymail@provider.com
○ (base) PS D:\Github\git_tests>
```

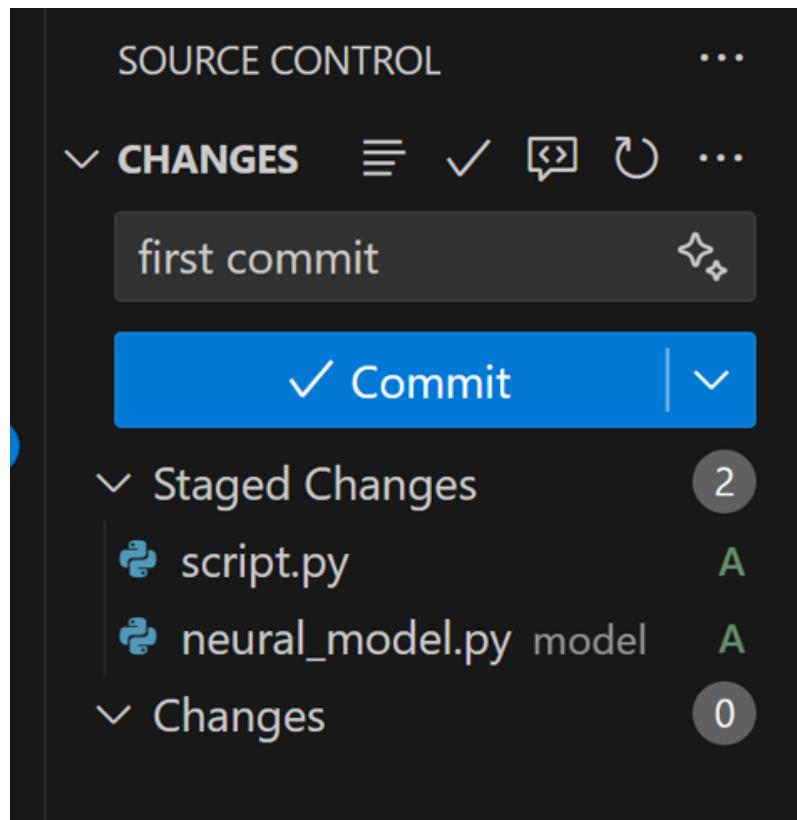
The terminal window shows two commands being run: 'git config user.name authorname' and 'git config user.email mymail@provider.com'. The first command is completed with a success status (●), while the second is still in progress (○).

Queste credenziali non devono essere per forza vere, serve solo a git per la stringa di commit. I cambiamenti possono essere fatti sul singolo file o su tutti:



Così facendo tutti i cambiamenti vengono messi in stage area. Vedremo quindi di fianco al file il simbolo **A** che indica che il file è stato aggiunto alla staging area. A questo punto dovremo fare il commit, per farlo dobbiamo scrivere un messaggio che descriva il cambiamento.

↓



Si creerà un graph che ci permette di vedere la lista concatenata dei commit che abbiamo fatto.



The screenshot shows the VS Code interface with the title bar "git\_tests". The left sidebar has "SOURCE CONTROL" expanded, showing a "CHANGES" section with a message "Message (Ctrl+Enter t...)" and a "Publish Branch" button. Below it, there's a commit history: "first commit" by "authorname" at "now (May 6, 2025 at 12:13 PM)". It shows "2 files changed, 20 insertions(+)" and a commit hash "6d7ea44". The main editor area contains a single line of Python code: "print("MLOps example")". The status bar at the bottom indicates "Ln 1, Col 23" and "Python".

In ciascuno di questi commit c'è un identificativo unico che è l'hash, che ricordiamo è un valore esadecimale che rappresenta in modo univoco il commit. Proviamo ad aggiungere delle modifiche al nostro file:

The screenshot shows the VS Code interface with the title bar "git\_tests". The left sidebar has "EXPLORER" expanded, showing a folder "GIT\_TESTS" containing "model", "neural\_model.py", and "script.py". The "script.py" file is selected and has a blue line above it, indicating a modification. The main editor area shows the following code in "script.py":

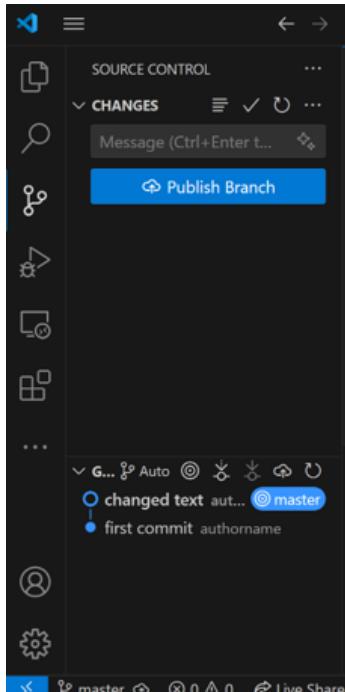
```
1 print("MLOps course, changed line 1")
2 print("Added a line")
3 |
```

The terminal below shows the command-line history:

```
(base) PS D:\Github\git_tests> git config user.name authorname
(base) PS D:\Github\git_tests> git config user.email mymail@provider.com
(base) PS D:\Github\git_tests> [ ]
```

The status bar at the bottom indicates "Ln 3, Col 1" and "Python".

Notiamo subito i cambiamenti nell'explorer di VSC, prima cosa vediamo una linea blu su VSC con le modifiche, facendoti vedere anche com'era prima, è passato poi da U ad M essendo stato modificato. Passo successivo lo mettiamo in stage area, cliccando sul simbolo + e poi facciamo il commit.



Notiamo subito che si è spostato il master, vuol dire che adesso la cosiddetta **HEAD** del nostro progetto è il master.

## 5.7 VSC + Git - Collaboration

Collaborare con altri sviluppatori è una delle cose più importanti quando si sviluppa un progetto. Git ci permette di farlo in modo semplice e veloce. Per pushare la repository dobbiamo cliccare sul pulsante **Publish Branch**, decidendo se renderla pubblica o privata. Una volta pubblicata su GitHub, nella sezione **Settings** possiamo aggiungere altri collaboratori, che potranno accedere alla nostra repository e fare modifiche.

The screenshot shows the GitHub repository settings page for a private repository. On the left, there's a sidebar with sections like General, Access (Collaborators selected), Code and automation (Branches, Tags, Rules, Actions, Webhooks, Environments, Codespaces, Pages), and Security (Advanced Security, Deploy keys, Secrets and variables). The main area is titled 'Collaborators and teams' and shows 'Private repository' status with a note 'Only those with access to this repository can view it'. It also shows 'Direct access' with a note '0 collaborators have access to this repository. Only you can contribute to this repository.' At the bottom, there's a 'Manage access' section with a note 'You haven't invited any collaborators yet' and a 'Add people' button.

Sempre su GitHub una volta fatto il commit vedremo i vari hash, chi l'ha generato ed altre informazioni che vedremo in seguito.

## 5.8 Branches

Sono dei set isolati di commit, che ci permettono di lavorare in maniera separata con il codice. Idealmente il branch serve da feature, ovvero per sviluppare una nuova funzionalità senza intaccare il codice principale. Questo è molto importante per fare bug fixing, code refactoring, etc.

## 5.9 Merging

Fatto il mio branch, dobbiamo rimettere "in ordine" il codice nel nostro tree, questo può essere fatto in automatico se non abbiamo fatto modifiche contemporanee con altre persone mandandole in conflitto, oppure sono state fatte sullo stesso file ma in punti diversi.

Oppure si fa manualmente:

- Modifico i file che hanno conflitto.
- Si rimette in stage area.
- Si fa un nuovo commit.
- Si riprova il merge.

Tutto questo va fatto ripetutamente finché non si risolvono tutti i conflitti.

# 6 DevOps vs MLOps

**DevOps** è un paradigma per cui si fa molto rapidamente lo sviluppo, il rilascio del codice ed il controllo di produzione. Generalmente si divide in 2 parti:

1. **Development (sviluppo):** dove si fa la pianificazione del codice, si affronta il problema e capisco come risolverlo. Scrivo il codice, faccio la build ed infine faccio i test.
2. **Ops:** in questa fase si fa il rilascio, decidendo come distribuirlo e su quale piattaforma. Facendo monitoring e dicendo come farlo funzionare.

**MLOps** è un po' diverso, abbiamo sempre il DevOps come parte integrante dello sviluppo di un'applicazione. Differisce però perchè che c'è una parte in più che è il Machine Learning. Nel **Machine Learning** la parte importante sono i **dati** ed il **modello**. Concettualmente sono molto simili ma ci sono delle differenze nei punti chiave, per esempio nel deployment abbiamo, nel DevOps abbiamo del codice che una volta che viene eseguito e gestito nell'ambito di sviluppo, verrà wrappato e in un eseguibile e deve essere validato su dei test (unit/integration). Nell'MLOps abbiamo un elemento in più nel Development che è la gestione e la creazione del modello di

Machine Learning per l’artefatto<sup>1</sup> finale. Qui la validazione viene fatto sui dati, per vedere come performa il modello, questo generalmente va fatto mentre si sviluppa il software e non durante la consegna al cliente. Il **Version Control** tra DevOps ed MLOps differisce perchè:

- Nel **DevOps** abbiamo → codice e l’artefatto/eseguibile generato.
- Nell’ **MLOps** abbiamo → abbiamo comunque del codice, abbiamo però in più la gestione degli iperparametri, la gestione delle metriche, i dati train/test ecc.

I concetti chiave del **DevOps** sono:

- **Continuous integration (CI)**: Quando il codice è stato scritto ed è in una versione funzionante, automaticamente vengono eseguiti tutti quanti gli unit test del caso, per verificare che il mio codice sia consistente e che non abbia rotto niente di quello che c’era prima. ← **Importante!**
- **Continuous delivery (CD)**: Fatto e testato il codice, una volta che tutto funziona lo mando in deploy, come faccio a farlo? C’è una parte del codice che dice che, ogni volta che ho controllato che tutto funzioni, manda tutto ad un server remoto mettendosi da solo in produzione.
- **Microservices**: Questo è un concetto, tendenzialmente vorrei lavorare con dei software che sono dei microservizi, che non hanno nessuna dipendenza. (Magari vogliamo che ogni servizio del backend sia un microservizio separato quindi, una parte con autenticazione, una parte che gestisca una cosa e via dicendo).
- **Infrastructure as Code (IaC)**: Sono tutti i sistemi definiti lato codice con degli script che costruiscono l’intero stack. (esistono software tipo Terraform che ci aiutano in questo)
- **Monitoring** and Instrumentation: Serve per prendere decisioni per quanto riguarda l’affidabilità e la sicurezza del codice.

## 6.1 Passaggio dal DevOps al MLOps

Prima di partire con il Machine Learning dobbiamo capire bene il DevOps (base di partenza da cui costruiamo tutto il resto). Nella pratica questo si traduce in implementazione di **Continuous integration** e **Continuous delivery**, tutte le altre parti sono molto meno critici, però esistono ed è bene sapere come vengono affrontati.

---

<sup>1</sup>Un artefatto, in questo contesto, è il risultato finale del processo di sviluppo: può essere un modello addestrato, un pacchetto software, uno script o qualsiasi altro prodotto generato dal workflow.

### 6.1.1 Continuous Integration

Voglio fare automaticamente test di integrazione o test di unità. Per esempio voglio vedere il lint<sup>2</sup> o il test, ecc. Ecco un esempio:

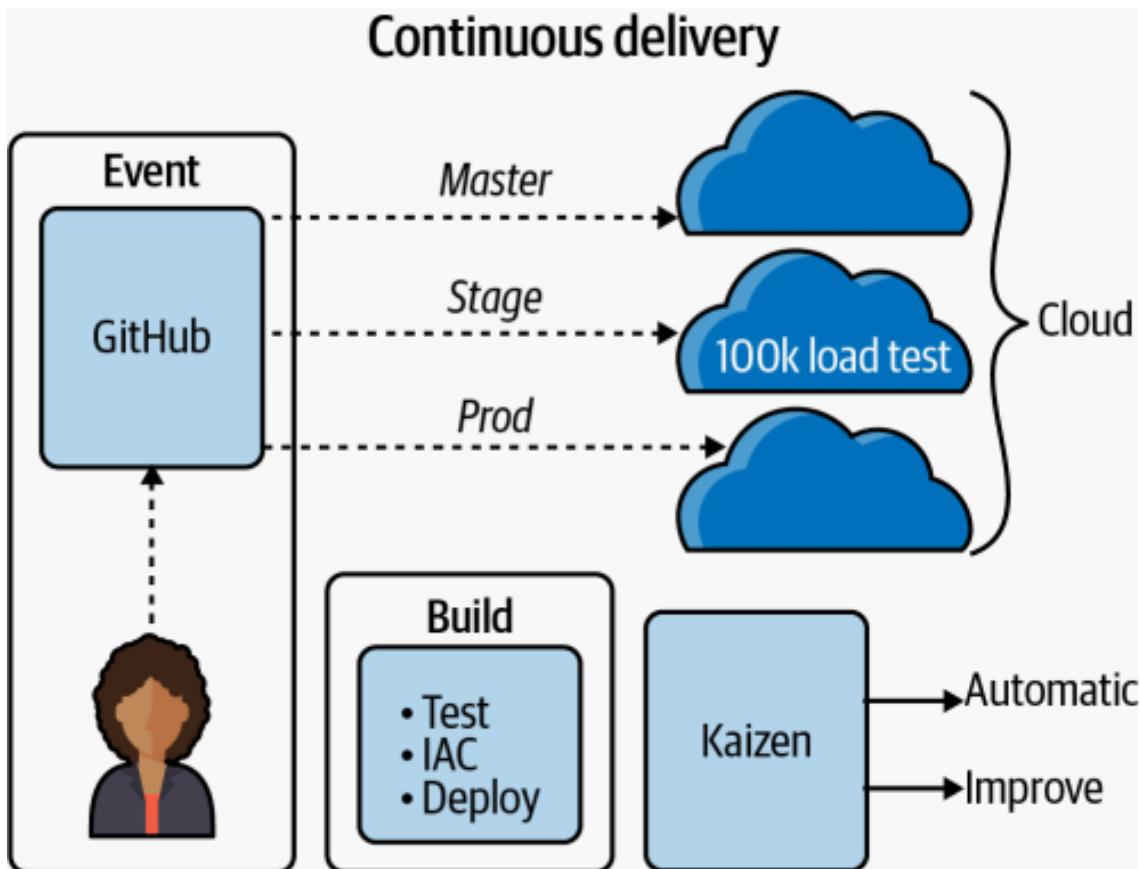
```
name: Deploy Python 3.5
on: [Push]
jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - name: Set up Python 3.12
        uses: actions/setup-python@v5
        with:
          python-version: '3.12'
      - name: Install dependencies
        run: |
          make install
      - name: Lint
        run: |
          make lint
      - name: Test
        run: |
          make test
```

---

<sup>2</sup>Python utilizza il duck-typing, cioè il tipo di una variabile viene determinato dal suo comportamento e non dalla sua dichiarazione esplicita. Questo può portare a errori difficili da individuare. Il **lint** è uno strumento che analizza il codice sorgente per individuare errori di sintassi, stile o potenziali bug, aiutando a mantenere il codice più pulito e conforme alle best practice.

### 6.1.2 Continuous Delivery

Una persona scrive un pezzo di codice, lo manda su GitHub. Nel momento in cui faccio subito un commit GitHub farà partire da solo tutto ciò che serve per fare delivery ↓



Quindi manda in produzione, carica un po' di volte il test per fare delle prove gestisce la parte cloud e via dicendo. Gestisce lui le cose automaticamente le cose (in base a quanto si paga).

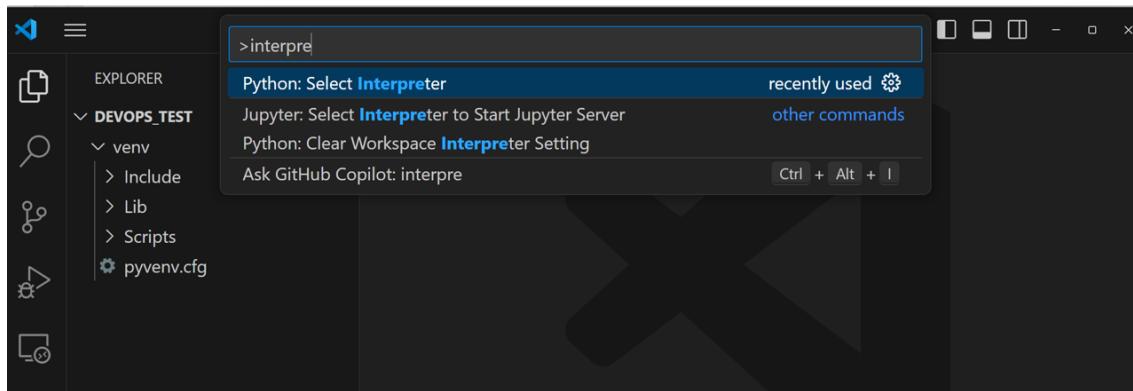
## 6.2 MLOps

MLOps è un'estensione del DevOps, l'unica cosa che dovremo fare in più è gestire la parte di ML. In un modo molto pratico MLOps funziona in questo modo:

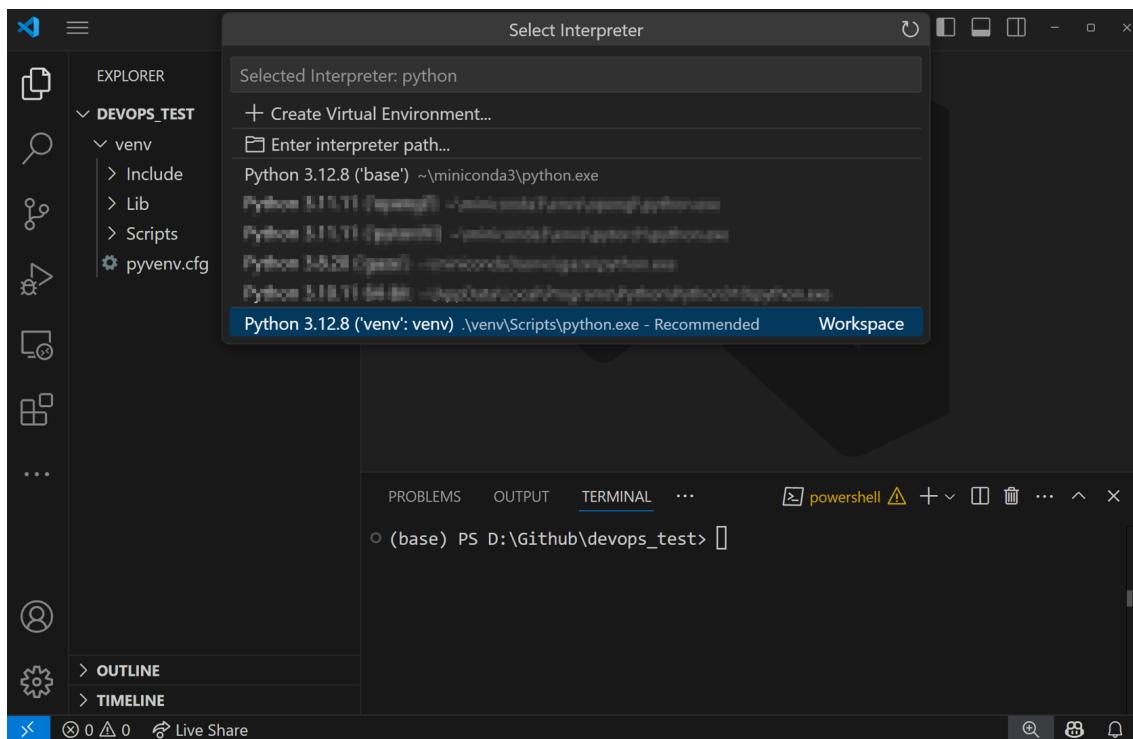
Prendo il codice, faccio il deploy e lo addestro, vedo come va il modello e se è funzionante, se non funziona riaddestro il modello e ricomincio. Tutto questo viene fatto in loop finché non è funzionante.

In MLOps c'è un 25% importante che riguarda tutta la struttura della repository, c'è poi un'altra parte importante che è il business (nel nostro progetto naturalmente non ci sarà), abbiamo poi un 25% di dati, che sono fondamentali per il nostro lavoro, tante volte questi dati dobbiamo saperli ricavare da soli. Abbiamo in fine un 25% di modello, la scelta ed il design del modello sono molto importanti. La prima cosa

da fare è installare un virtual environment (qui verrà spiegato con venv, metodo più didattico, nel progetto useremo miniconda). Iniziamo digitando sul terminale (che ricordiamo si apre con **ctrl+j** su VSC), e digitiamo **python -m venv venv**. Una volta fatto ciò (dopo aver installato l'estensione per python su VSC), schiacciano **ctrl+p** si aprirà una schermata in cui digitare **>interpreter**.



Selezionando **Python: Select Interpreter** si aprirà questa tendina dove selezioniamo **Workspace**.



Questa cartella ovviamente non la vorremo sul Source Control. Dobbiamo poi controllare se l'interprete che abbiamo caricato sia quello corretto, si può fare in 2 modi:

- **Linux:** (Sul terminale) `which "nome_eseguibile"` → ci dirà dove si trova quell'eseguibile nel sistema operativo.
- **Windows:** (Sul terminale) `where.exe "nome_eseguibile"` → dovessero esserci più risultati sappiamo che il primo è quello usato principalmente.

Per attivare manualmente l'interprete manualmente i comandi da terminale sono:

- **Linux/macOS:** `source venv/bin/activate`
- **Windows (PowerShell):** `.\venv\Scripts\Activate.ps1`
- **Windows (CMD):** `venv\Scripts\activate.bat`

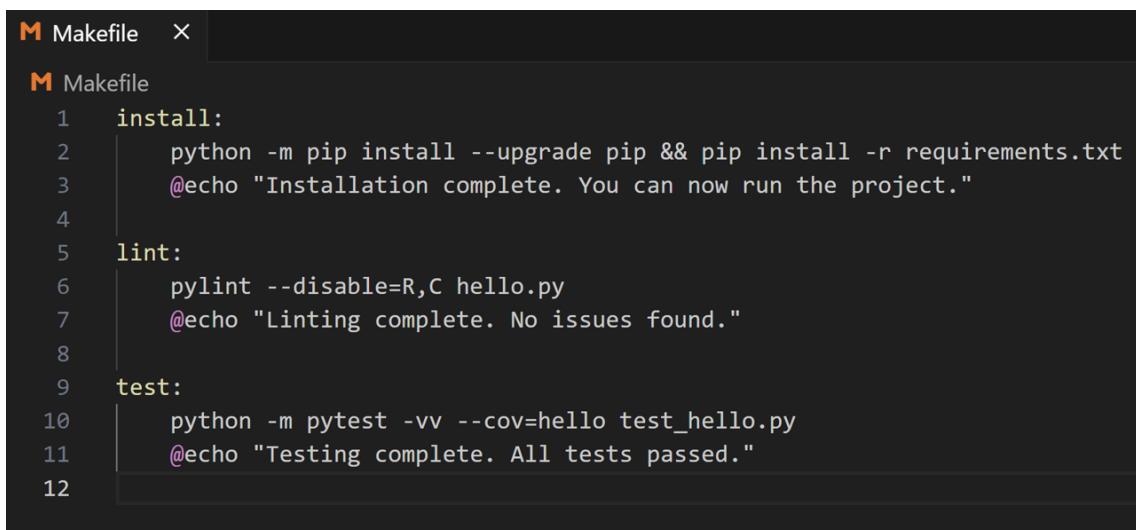
## 6.3 Esempio (e corso accelerato di Python)

Definiamo una repo GitHub in questo modo:

- MakeFile
- requirements.txt
- hello.py
- test\_hello.py
- venv

### 6.3.1 MakeFile

Utilissimo per velocizzare i processi di produzione. Ecco un esempio di MakeFile



```
M Makefile  X
M Makefile
1 install:
2     python -m pip install --upgrade pip && pip install -r requirements.txt
3     @echo "Installation complete. You can now run the project."
4
5 lint:
6     pylint --disable=R,C hello.py
7     @echo "Linting complete. No issues found."
8
9 test:
10    python -m pytest -vv --cov=hello test_hello.py
11    @echo "Testing complete. All tests passed."
12
```

Il Makefile contiene tre target che automatizzano operazioni di sviluppo Python:

**1. install:** Aggiorna pip e installa le dipendenze

```
python -m pip install --upgrade pip && pip install -r requirements.txt
@echo "Installation complete.
You can now run the project."
```

- `python -m pip install --upgrade pip`: Aggiorna pip all'ultima versione
- `pip install -r requirements.txt`: Installa tutti i pacchetti specificati nel file `requirements.txt`
- `@echo`: Mostra un messaggio di conferma (il simbolo `@` evita di stampare il comando stesso)

**2. list:** Esegue il linting del codice

```
pylint --disable=R,C hello.py
@echo "Linting complete. No issues found."
```

- `pylint`: Strumento di analisi statica per Python
- `--disable=R,C`: Disabilita i controlli per:
  - `R` (Refactor suggestions)
  - `C` (Convention violations)
- Analizza specificamente il file `hello.py`

**3. test:** Esegue i test unitari con coverage

```
python -m pytest -vv --cov=hello test_hello.py
@echo "Testing complete. All tests passed."
```

- `python -m pytest`: Esegue pytest come modulo
- `-vv`: Modalità verbose (doppia verbosità)
- `--cov=hello`: Misura la coverage del modulo `hello`
- `test_hello.py`: Esegue i test contenuti in questo file

## Note importanti

- Nei Makefile, i comandi devono usare tabulazioni (non spazi) per l'indentazione
- Il target `1 install` è sintatticamente errato (i nomi non possono contenere spazi)
- I separatori `--` non sono standard nei Makefile

Per eseguire un MakeFile dobbiamo digitare questi comandi sul terminale:

- Linux: → `sudo apt update && sudo apt install make`
- Windows: → `winget install -e -id GnuWin32.Make`
- Windows: → `choco install make`

Facciamo poi **make install**, il quale aggiorna pip all'ultima versione ed automaticamente fa tutto ciò che c'è scritto.

```
● (base) PS D:\Github\devops_test> make install
python -m pip install --upgrade pip && pip install -r requirements.txt
Requirement already satisfied: pip in d:\github\devops_test\venv\lib\site-packages (25.1.1)
Collecting pylint (from -r requirements.txt (line 1))
  Downloading pylint-3.3.7-py3-none-any.whl.metadata (12 kB)
Collecting pytest (from -r requirements.txt (line 2))
  Downloading pytest-8.3.5-py3-none-any.whl.metadata (7.6 kB)
Collecting pytest-cov (from -r requirements.txt (line 3))
  Downloading pytest_cov-6.1.1-py3-none-any.whl.metadata (28 kB)
Collecting astroid<=3.4.0.dev0,>=3.3.8 (from pylint->-r requirements.txt (line 1))
  Downloading astroid-3.3.9-py3-none-any.whl.metadata (4.5 kB)
Collecting colorama>=0.4.5 (from pylint->-r requirements.txt (line 1))
  Using cached colorama-0.4.6-py2.py3-none-any.whl.metadata (17 kB)
```

```
Collecting packaging (from pytest->-r requirements.txt (line 2))
  Downloading packaging-25.0-py3-none-any.whl.metadata (3.3 kB)
Collecting pluggy<2,>=1.5 (from pytest->-r requirements.txt (line 2))
  Downloading pluggy-1.5.0-py3-none-any.whl.metadata (4.8 kB)
Collecting coverage>=7.5 (from coverage[toml]>=7.5->pytest-cov->-r requirements.txt (line 1))
  Downloading coverage-7.8.0-cp312-cp312-win_amd64.whl.metadata (8.7 kB)
Downloading pylint-3.3.7-py3-none-any.whl (522 kB)
Downloading astroid-3.3.9-py3-none-any.whl (275 kB)
Downloading isort-6.0.1-py3-none-any.whl (94 kB)
Downloading mccabe-0.7.0-py2.py3-none-any.whl (7.3 kB)
Downloading pytest-8.3.5-py3-none-any.whl (343 kB)
Downloading pluggy-1.5.0-py3-none-any.whl (20 kB)
Downloading pytest_cov-6.1.1-py3-none-any.whl (23 kB)
Using cached colorama-0.4.6-py2.py3-none-any.whl (25 kB)
Downloading coverage-7.8.0-cp312-cp312-win_amd64.whl (214 kB)
Downloading dill-0.4.0-py3-none-any.whl (119 kB)
Downloading platformdirs-4.3.7-py3-none-any.whl (18 kB)
Using cached tomllib-0.13.2-py3-none-any.whl (37 kB)
Downloading iniconfig-2.1.0-py3-none-any.whl (6.0 kB)
Downloading packaging-25.0-py3-none-any.whl (66 kB)
Installing collected packages: tomllib, pluggy, platformdirs, packaging, mccabe, isort
Successfully installed astroid-3.3.9 colorama-0.4.6 coverage-7.8.0 dill-0.4.0 iniconfig-2.1.0 pylint-3.3.7 pytest-8.3.5 pytest-cov-6.1.1 tomllib-0.13.2
Installation complete. You can now run the project.
>(base) PS D:\Github\devops_test> []
```

A questo punto facciamo il test **make lint**:

```

● (base) PS D:\Github\devops_test> make lint
pylint --disable=R,C hello.py

-----
Your code has been rated at 10.00/10

Linting complete. No issues found.
✿ (base) PS D:\Github\devops_test>

```

Facciamo poi il **make test**:

```

(base) PS D:\Github\devops_test> make test
python -m pytest -vv --cov=hello test_hello.py
===== test session starts =====
platform win32 -- Python 3.12.8, pytest-8.3.5, pluggy-1.5.0 -- D:\Github\devops_test\venv\Scripts\python.exe
cachedir: .pytest_cache
rootdir: D:\Github\devops_test
plugins: cov-6.1.1
collected 1 item

test_hello.py::test_add PASSED

===== tests coverage =====
coverage: platform win32, python 3.12.8-final-0 __

      Name      Stmts   Miss  Cover
----- 
hello.py        3      0   100%
----- 
TOTAL          3      0   100%
===== 1 passed in 0.05s =====
Testing complete. All tests passed.
✿ (base) PS D:\Github\devops_test>

```

### 6.3.2 Requirements.txt

È una lista di nomi di package.

```

pylint
pytest
pytest-cov

```

Questi sono quelli fondamentali per questo test.

### 6.3.3 Test script

Semplice script:

```

def add(x: int, y: int) -> int:
    """Add two numbers."""
    return x+y
print(add(1,2)) #Output:3

```

```
from hello import add

def test_add():
    assert add(1, 2) == 3
    assert add(-1, 1) == 0
    assert add(0, 0) == 0
    assert add(-1, -1) == -2
    assert add(1000000, 2000000) == 30000000
```

Tutti le funzioni che testano un metodo devono chiamarsi `test_` e poi il nome, altrimenti non vengono riconosciuti.

Il gitignore di questo esempio sarà:

```
.pytest_cache/
venv/
.coverage

*.pyc
--pycache__/
*.pyo
*.pyd
```

### 6.3.4 Continous Integration

Ora dobbiamo creare le seguenti cartelle: .github → workflow → file chiamato **deploy.yml**

```
! deploy.yml u ×

.github > workflows > ! deploy.yml

1   name: Deploy Python 3.5
2   on: [push]
3   jobs:
4     build:
5       runs-on: ubuntu-latest
6       steps:
7         - uses: actions/checkout@v4
8         - name: Set up Python 3.12
9           uses: actions/setup-python@v5
10          with:
11            python-version: '3.12'
12          - name: Install dependencies
13            run: |
14              make install
15          - name: Lint
16            run: |
17              make lint
18          - name: Test
19            run: |
20              make test
21
```

Spieghiamo brevemente:

- name: → semplicemente il nome che gli diamo noi
- on: [push] → faccio un commit e poi fa il push su github e lo esegue
- runs-on: ubuntu-latest → esegue su sistema operativo ubuntu
- uses: action/checkout@v4 → scarica la repository
- name: Set up Python 3.12 → scarica python in quella versione e si crea l'ambiente

- quelle sotto sono le dipendenze fatte invece automatico dal MakeFile

Una volta creato questo file faccio Publish Branch, il commit della action lo vediamo sotto actions nella nostra pagina GitHub (chiamata build).

## 7 Data Science and EDA

Il Machine Learning parte dai dati, dobbiamo quindi essere sicuro di cosa parlano i dati che stiamo trattando, utilizzando male i dati potrebbero portare ad un modello funzionante ma con una comprensione del mondo esterna sbagliata. **Data Science**

- Tutto in AI parte dai dati, senza di questi non possiamo addestrare niente. I modelli di AI prendono un esempio, facendo un'ipotesi e andando a tentativi finché non trovano quella corretta. Continuano poi a lavorare su quella corretta fino ad arrivare a convergenza.
- Questi modelli sono **Statistici**, per lavorare bene nell'ambito del machine learning la comprensione statistica è fondamentale.
- I dati hanno una forma (rappresentazione), una buona rappresentazione per i dati naturali è la Gaussiana. I modelli imparano dai dati (imparano una sorta di rappresentazione statistica).
- Se ho dei dati di scarsa qualità (es: introduzione di un bias senza saperlo), avremo, come detto prima, un modello pessimo.

Quando non abbiamo i dati possiamo quindi ricadere in alcuni problemi:

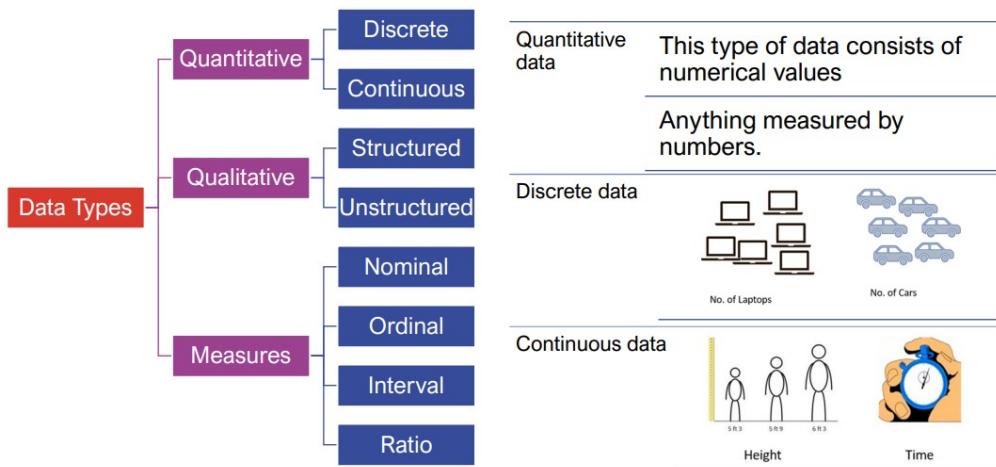
- Rumore quindi notazioni poco corrette.
- Pochi dati perchè magari alcune classi sono poco rappresentate (a volte è un limite economico). → Sottorappresentazione
- I Biases possono crearcì diverse tipologie di problemi, che a volte, senza l'acquisizione di altri dati, sono irrisolvibili.
- Possiamo avere un dataset sbilanciato, il che porta ad avere dei dati poco o troppo rappresentati.
- Ce ne sono anche altri di cui non parleremo.

Data Science Workflow:

- **Input:** si parte naturalmente dai dati.
- **Data Cleaning:** prendiamo i dati e li controlliamo TUTTI, verificando che non ci siano errori (annotazioni sbagliate, immagini rotte, ecc). Lo si fa per tutti ma non uno per uno, lo faremo con metodi statistici.

- **Data Representation:** dobbiamo capire come rappresentare i dati, ad esempio tante immagini o tabelle con migliaia di colonne... Questa rappresentazione va bene per il Machine Learning (tirando fuori le informazioni che sappiamo essere importanti per la risoluzione del nostro problema), di meno per il Deep Learning.
- **Preprocessing:** si occupa di scalare i nostri dati (che si trovano in un dominio numerico di qualunque tipo) e li dobbiamo portare in qualcosa di interpretabile facilmente dal mio sistema (in un sistema di Machine Learning si dice che normalizzo i dati rispetto ad un intervallo).
- **Model Training:** divido il mio modello in training e testing:
  - **training:** il modello prende i dati, ci lavora sopra dandoci in output un risultato.
  - **testing:** questa parte non dovrà mai essere vista dal modello in training, vengono usati solo per capire come potrebbe performare nella realtà.
- **Performance Measurements:** alla fine dovremo misurare statisticamente le performance, in base ai risultati capiamo se il modello ha imparato bene o meno le informazioni date tramite i dati.

## 7.1 Data types: quantitative



Ci sono diverse tipologie di dati e vengono trattati in modo diversi:

- **Quantitativi:** Sono i dati che posso misurare ed alla quale posso attribuire un valore.
  - Continui: Sono dati "infiniti", altezza, orario ...
  - Discreti: Numeri interi o comunque finiti, con una certa separazione nello spazio in cui vivono.

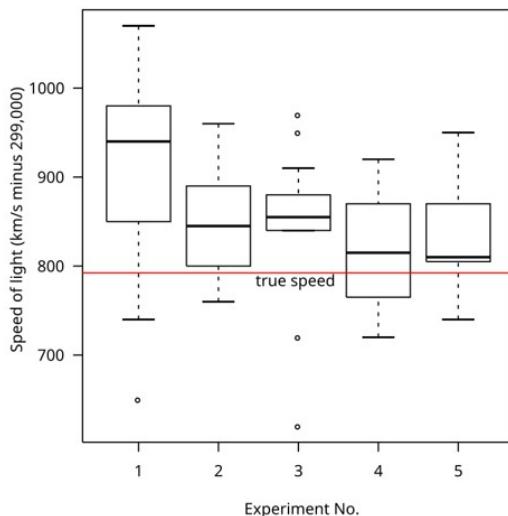
- **Qualitativi:** Sono dati che non sono direttamente rappresentabili tramite misurazioni.
  - Strutturati: Ad esempio tabelle, stringhe ...
  - Non Strutturati: Dati di partenza tipo sorgenti testuali, codice delle applicazione ...
- **Misure:**
  - Nominali: Ad esempio maschio/femmina, tutte cose categorizzabili.
  - Ordinali: Ad esempio la taglia di una maglia ...
  - Intervalli: Ad esempio temperatura misurata in gradi celsius ...
  - Di rapporto: Uguali agli intervalli ma hanno lo zero assoluto.

## 7.2 Exploratory Data Analysis (EDA)

Questa parte si occupa di capire i dati che stiamo utilizzando, tramite strumenti di visualizzazione principalmente qualitativi (visualizzazione quindi grafici ecc.) o quantitativi (enumarazione). I dati generalmente sono sporchi (perchè costa molto sistemare i dati), per questo è molto importante un'analisi precisa dei nostri dati e di come utilizzarli. Gli strumenti principali di visualizzazione sono:

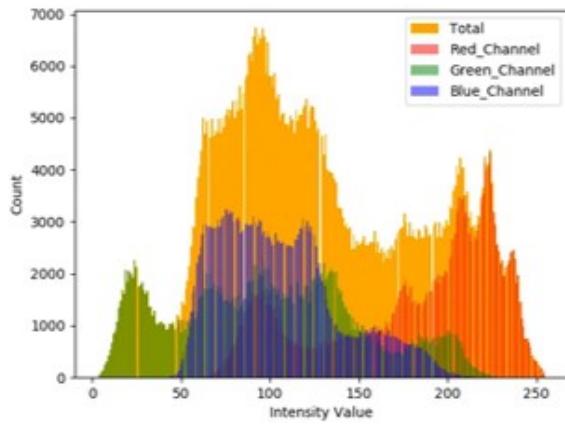
- **Graphical Tools:**

- **Box Plot:** Strumento grafico per mostrare una di certa variabile la sua località, la sua distribuzione, quanto può essere sbilanciata rispetto ad un certo valore e quindi avere tramite quartili una rappresentazione del dato più uniforme possibile.

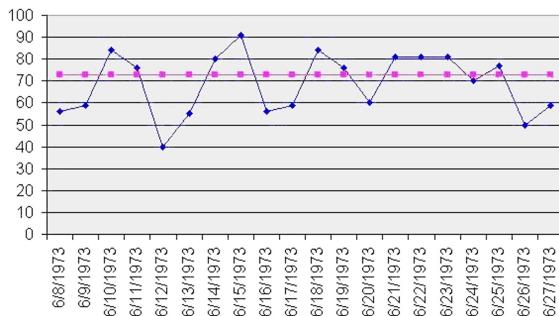


- **Histogram:** Strumento di visualizzazione della distribuzione di dati quantitativi, nell'immagine sottostante vediamo la distribuzione della

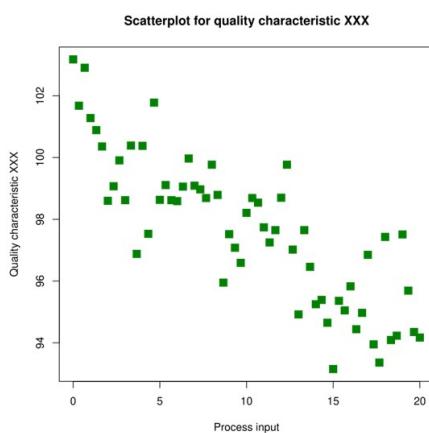
codifica del colore del pixel. Si può usare per altre cose ad esempio le variabili categoriche o altri tipi di variabili in cui è utile questo tipo di visualizzazione grafica.



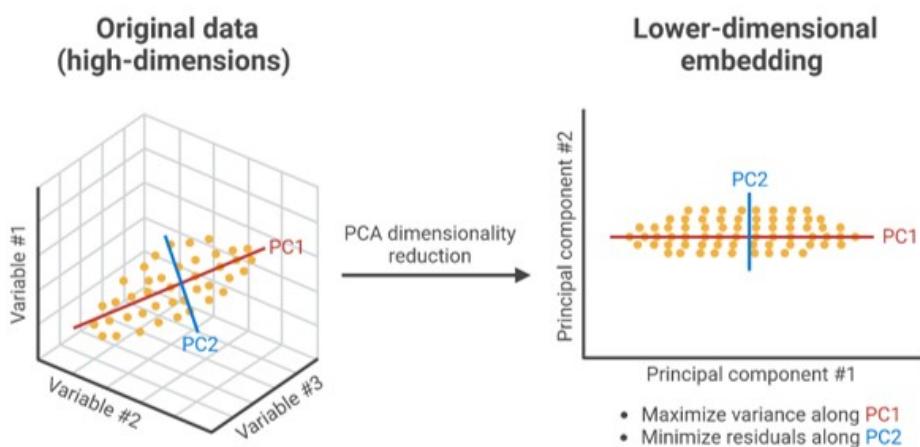
- **Run Chart:** È utile se abbiamo delle serie temporali, si riesce a fare un’analisi di come potrebbe essere il nostro dato lungo una variabile indipendente.



- **Scatter Plot:** Nella quale decidiamo 2 variabili e si fa un plot di quest’ultime lungo un piano cartesiano. Non ci fa vedere solo la quantità delle 2, ci fa vedere il loro rapporto, come evolvono, in pratica mette in relazione tra di loro i dati correlati.



Una volta che ho rappresentato i miei dati, una cosa che può risultare utile è la riduzione della dimensionalità. Alcuni dati potrebbero essere difficili da rappresentare (ad esempio ha più di 3 dimensioni), quello che possiamo, utilizzando gli stessi strumenti di prima, ridurre la dimensionalità e passare magari da 4 dimensioni ad 1. Lo faccio tramite una tecnica chiamata **PCA**. Lo scopo di questa tecnica è: prendo la dimensione massima di massima diffusione (spread) del mio dato, e da questo vado ad estrarre i più alti autovalori della matrice di covarianza e ottengo PC1, PC2. Una volta che le ho posso tagliare le misure che non sono di principale rilevanza, tenendo quindi solo quelle di principale estensione.



Ricordiamoci che ogni ad ogni informazioni che tagliamo perdiamo precisione quindi questo processo va fatto con cautela e solo quando serve. Ad esempio quando prendiamo dei dati che possono essere intuitivi per noi ad alte dimensioni, ad esempio una scatola (altezza - larghezza - volume - profondità) e la trasformo soltanto in 2 dimensioni che in quel momento possono esserci utili, ad esempio il peso e magari la profondità.

## 7.3 Preprocessing in DL

Il preprocessing nelle reti neurali è abbastanza limitato, questo perchè l'estrazione delle feature e le informazioni principali e più importanti del mio dato lo fa direttamente il modello quando viene addestrato. Un sistema di Deep Learning funziona così:

1. Prendo un dato.
2. Da questo dato estraggo le informazioni più importanti di questo dato.
3. Dalle informazioni faccio la classificazione o il mio task finale.

Che informazioni estraggo? Ad esempio da un'immagine come faccio a sapere cosa devo andare a guardare? Lo sa già il modello, non glielo dobbiamo dire noi. Con il preprocessing noi non facciamo molto, solitamente è solo un problema strettamente numerico, tutto il resto è automatico.

## 7.4 Model Training & Selection

I sistemi di deep learning si addestrano con un framework, noi useremo Pytorch che è quello che da più controllo in assoluto tra quelli disponibili. Partiamo con un semplice script di python ↓

```
"""
Install requirements with:
pip install torch torchvision scikit-learn matplotlib seaborn tqdm
"""

import time
import torch
import torch.nn as nn
import torchvision
import torchvision.transforms as transforms
from tqdm import tqdm
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.metrics import classification_report, confusion_matrix
# Automatically download and load MNIST data
transform = transforms.Compose([transforms.ToTensor()])
train_set = torchvision.datasets.MNIST(root='./data', train=True,
    download=True, transform=transform)
test_set = torchvision.datasets.MNIST(root='./data', train=False,
    download=True, transform=transform)

train_loader = torch.utils.data.DataLoader(train_set, batch_size
    =64, shuffle=True)
test_loader = torch.utils.data.DataLoader(test_set, batch_size
    =1000, shuffle=False)
```

↑ Questo codice configura l'ambiente per addestrare e valutare un modello di machine learning sul dataset MNIST (numeri scritti a mano) utilizzando PyTorch. Ecco una spiegazione dettagliata:

- Installazione delle dipendenze:

```
pip install torch torchvision scikit-learn matplotlib
seaborn tqdm
```

Installa i pacchetti necessari:

- torch: Libreria principale per il deep learning.
- torchvision: Dataset e strumenti per computer vision.
- scikit-learn: Metriche di valutazione.
- matplotlib e seaborn: Visualizzazione dati.
- tqdm: Barre di avanzamento.

- Importazione delle librerie:

```
import time
import torch
import torch.nn as nn
import torchvision
import torchvision.transforms as transforms
from tqdm import tqdm
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.metrics import classification_report,
confusion_matrix
```

Importa tutti i moduli necessari per:

- Gestione tensori e reti neurali (torch, nn)
- Caricamento dati (torchvision)
- Preprocessing immagini (transforms)
- Visualizzazione (plt, sns)
- Valutazione del modello (classification\_report, confusion\_matrix)

- Processing dei dati:

```
transform = transforms.Compose([transforms.ToTensor()])
```

Crea una trasformazione che:

- Converte le immagini in tensori PyTorch
- Normalizza automaticamente i pixel in intervallo [0, 1]

- Caricamento del dataset MNIST:

```
train_set = torchvision.datasets.MNIST(
    root='./data',
    train=True,
    download=True, # Scarica se non presente
    transform=transform
```

```

        )

    test_set = torchvision.datasets.MNIST(
        root='./data',
        train=False,
        download=True,
        transform=transform
    )

```

- Training set: 60,000 immagini
- I dati vengono salvati in ./data
- Le immagini sono convertite in tensori automaticamente

- Creazione dei DataLoader:

```

train_loader = torch.utils.data.DataLoader(
    train_set,
    batch_size=64,    # 64 campioni per batch
    shuffle=True      # Mescola i dati ad ogni epoca
)

test_loader = torch.utils.data.DataLoader(
    test_set,
    batch_size=1000,   # Batch piu grandi per test
    shuffle=False     # Non mescolare per valutazione
)

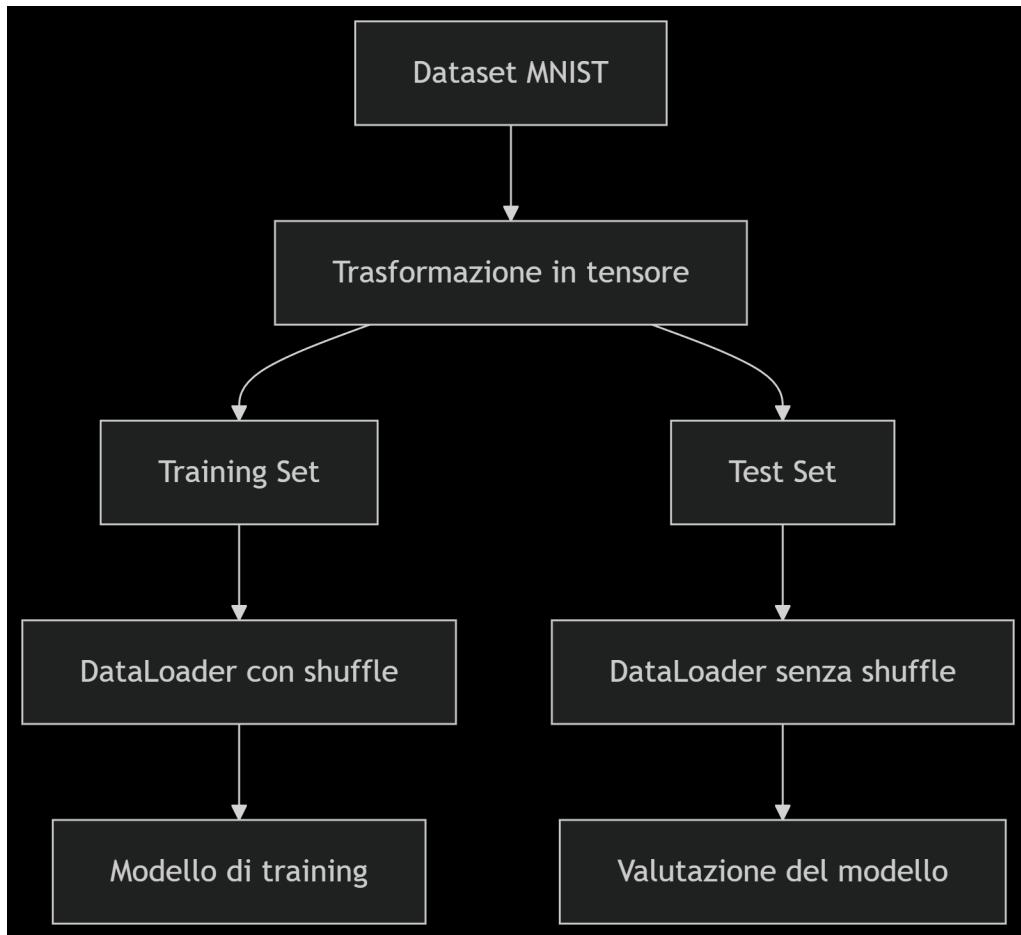
```

Funzionamento del DataLoader:

Table 1: Confronto DataLoader Training vs Test

Parametro	Training	Test
Batch size	64 (dimensione ottimale per l'aggiornamento dei pesi)	1000 (dimensione maggiore per valutazione più veloce)
Shuffle	Attivato (mescola i dati ad ogni epoca per prevenire overfitting)	Disattivato (mantiene l'ordine originale per valutazione consistente)
Funzione	Fornisce batch di dati al modello durante l'addestramento	

Struttura complessiva del flusso dati ↓



Facciamo ora un po' di EDA, il codice riportato sotto è un'analisi molto semplice per capire cosa vuol dire lavorare con un dataset nuovo e capire cosa fa:

```
# Exploratory Data Analysis (EDA)
print("EDA Reports:")
print("- Train data size:", len(train_set))
print("- Test data size:", len(test_set))

# Check distribution of classes
labels = train_set.targets.numpy()
unique, counts = np.unique(labels, return_counts=True)
class_dist = dict(zip(unique, counts))
print("- Class distribution:", class_dist)

# Plot class distribution
plt.figure(figsize=(10,4))
sns.barplot(x=list(class_dist.keys()), y=list(class_dist.values()))
plt.title("Class distribution in training set")
plt.xlabel("Digit")
plt.ylabel("Count")
```

```

plt. show()

# Show sample images
examples = enumerate(train_loader)
batch_idx, (example_data, example_targets) = next(examples)
plt.figure(figsize=(10,4))
for i in range(10):
    plt.subplot(2,5,i+1)
    plt.imshow(example_data[i][0], cmap='gray', interpolation='none')
    plt.title(f"Label: {example_targets[i].item()}")
    plt.axis('off')
    plt.tight_layout()

plt. show()

```

↑ Come detto prima questo codice esegue un'Analisi Esplorativa dei Dati (EDA) su un dataset di immagini (probabilmente MNIST o simile), utilizzando PyTorch e librerie di visualizzazione. Ecco una spiegazione dettagliata:

- Stampa delle dimensioni del dataset:

```

print("- Train data size:", len(train_set))
print("- Test data size:", len(test_set))

```

Mostra il numero di campioni nel training set e nel test set.

Output esempio:

- Train data size: 60000
- Test data size: 10000

- Analisi della distribuzione delle classi:

```

import numpy as np

labels = train_set.targets.numpy()
unique, counts = np.unique(labels, return_counts=True)
class_dist = dict(zip(unique, counts))
print("- Class distribution:", class_dist)

```

Questa parte di codice estrae le etichette (labels) del training set, conta le occorrenze di ogni classe (cifre da 0 a 9) e stampa la distribuzione in formato dizionario.

Output esempio:

- Class distribution: {0: 5923, 1: 6742, 2: 5958, 3: 6131, 4: 5842, 5: 5421, 6: 5918, 7: 6265, 8: 5851, 9: 5949}

- Grafico a barre della distribuzione:

```

import matplotlib.pyplot as plt
import seaborn as sns

plt.figure(figsize=(10, 6))

```

```

sns.barplot(x=list(class_dist.keys()), y=list(
    class_dist.values()))
plt.xlabel('Digit')
plt.ylabel('Count')
plt.title('Distribuzione delle classi nel dataset MNIST')
)
plt.show()

```

Questa parte genera un grafico a barre che mostra il conteggio dei campioni per ogni classe, usa seaborn per una visualizzazione chiara e dà le etichette “Digit” all’asse x e “Count” all’asse y. È utile perché identifica sbilanciamenti tra le classi.

#### 4. Visualizzazione di immagini campione:

```

import matplotlib.pyplot as plt

examples = enumerate(train_loader)
batch_idx, (example_data, example_targets) = next(
    examples)

```

Carica un batch di dati dal DataLoader del training set e prende il primo batch (di solito 64 o 128 immagini).

```

plt.figure(figsize=(12, 6))
for i in range(10):
    plt.subplot(2, 5, i+1)
    plt.imshow(example_data[i][0], cmap='gray')
    plt.title(f"Label: {example_targets[i].item()}")
    plt.axis('off')

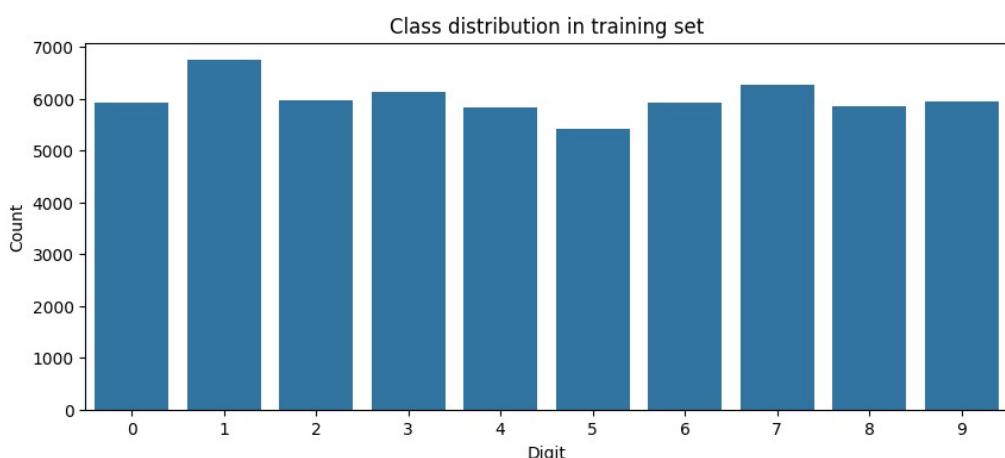
plt.tight_layout()
plt.show()

```

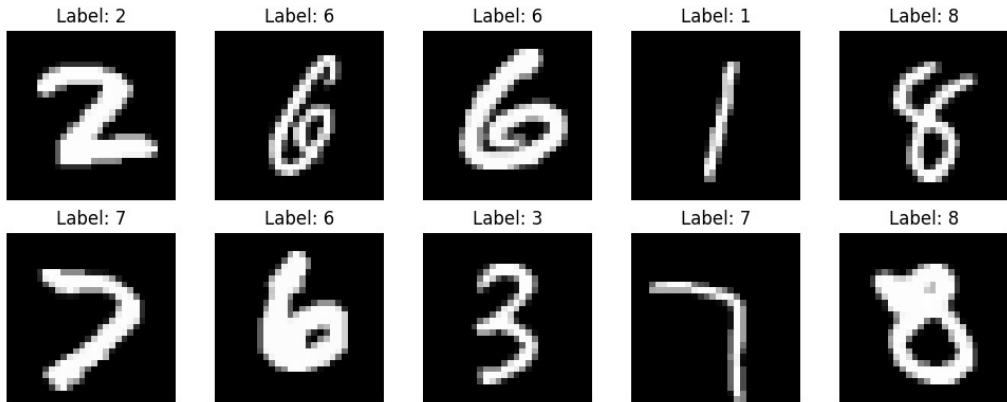
Questa parte infine crea una griglia 2x5 con 10 immagini, mostra ogni immagine in scala di grigi (`cmap='gray'`), aggiunge l’etichetta come titolo.

`example_data[i][0]` seleziona il primo canale (immagini in bianco e nero).

Facciamo vedere delle rappresentazioni visive del comportamento, ad esempio un possibile output della distribuzione può essere:



Da qui vediamo che ad esempio abbiamo un piccolo sbilanciamento della classe 1, notiamo comunque che è ben bilanciato. Andiamo a vedere quindi qualche esempio (10 elementi casuali):



Questo può essere utile perchè da qui possiamo vedere se c'è qualche classe strana, noi ad occhio essendo digits li riconosciamo subito, però risulta subito palese come l'1 sia un elemento più semplice del 3 o dell'8.

#### 7.4.1 Training

La parte di training è molto importante ed è ciò che succede sotto il lavoro di addestramento.

```
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)
```

↑ Quando si definisce un meccanismo di addestramento di reti neurali si fanno 3 cose:

1. Si definisce il modello su cui vai a fare il lavoro.
2. Si definisce il criterio, ovvero l'obbiettivo che il modello deve andare ad ottimizzare.
3. Si attua il meccanismo di **discesa stocastica del gradiente**, ovvero andiamo a computare con la nostra architettura la predizione che andiamo a fare sul nostro dato, andando a vedere quanto questo dato è discrepante dal risultato finale.

↓ Questo è un ciclo di training di una rete neurale (in particolare sui dati che abbiamo visto prima)

```
# Train the model on CPU
start = time.time()      #calcolo il tempo
for epoch in tqdm(range(5)):    #5 iterazioni, un'epoca e' un
    intero giro di addestramento sul dataset (1 volta)
    model.train()    #metto il modello in modalita' di training
    running_loss = 0.0
```

```

for images, labels in train_loader: #itero sul dataset composto
    da immagini e categoria
    optimizer.zero_grad()      #processo stocastico, deve
        resettare il gradiente
    outputs = model (images)    #faccio l'inferenza, do alla
        rete neurale l'input e lui restituisce un output (questo
        prende l'immagine e ritorna la classe)
    loss = criterion(outputs, labels)
    loss.backward()            #fa imparare il modello: prendo il
        modello, gli do un'immagine e lui ci dà una risposta.
        Confronto l'immagine col mio dato di riferimento, vedo
        quanto è largo l'errore, più sbagli meglio impara
    optimizer.step()          #finito il giro ne fa un altro
    running_loss += loss.item()
    print(f'Epoch [{epoch +1}/5], Loss: {running_loss/len(
        train_loader) :. 4f}')
print(f'Training time: {time.time() - start :. 2f} seconds')

```

Codice di valutazione del modello ↓

```

# Evaluate the model
model.eval()      #non sono più in training ma lo sto valutando
correct = 0
total = 0
all_preds = []
all_labels = []

with torch.no_grad():      #il loop sarà ora sul testing set
    for images, labels in test_loader:
        outputs = model(images)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()
        all_preds.extend(predicted.numpy())
        all_labels.extend(labels.numpy())

accuracy = correct / total
print(f'nTest accuracy: {accuracy :. 4f}')

```

↑ Questo blocco di codice mette il modello in modalità di valutazione (disabilitando dropout e batch-norm adattativi) e poi, senza calcolare gradienti, scorre tutto il set di test generando predizioni per ogni batch. Confronta le etichette previste con quelle vere per contare quante sono corrette, tiene traccia del numero totale di esempi e accumula in due liste le predizioni e le etichette reali (utile per calcoli di metriche più avanzate). Infine calcola l'accuratezza come rapporto tra esempi classificati correttamente e totale degli esempi, e la stampa in output.

Fatto questo posso calcolarmi il classification report e la matrice di confusione . $\Downarrow$

```
# Classification report
print("\nClassification Report:")
print(classification_report(all_labels, all_preds))

# Plot confusion matrix
conf_matrix = confusion_matrix(all_labels, all_preds)
plt.figure(figsize=(8,6))
sns. heatmap(conf_matrix, annot=True, fmt='d', cmap='Blues')
plt.xlabel('Predicted')
plt.ylabel('True')
plt.title('Confusion')
plt.show()
```

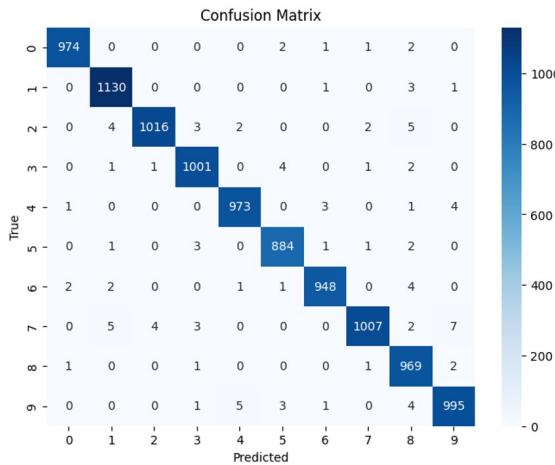
$\Updownarrow$  Questo pezzo di codice prende le liste delle etichette vere e delle predizioni raccolte durante la fase di valutazione e:

1. Genera un report statistico: tramite **classification\_report** di scikit-learn, stampa per ciascuna classe metriche come precisione, richiamo (recall), F1-score e supporto (numero di esempi), offrendoti un quadro dettagliato delle prestazioni del modello su ogni categoria.
2. Costruisce e visualizza la matrice di confusione: calcola con **confusion\_matrix** la tabella in cui le righe sono le etichette vere e le colonne quelle predette, quindi la disegna usando Seaborn. Il grafico mostra, cella per cella, il numero di occorrenze di ciascuna combinazione (vero positivo, falso negativo ecc.), con annotazioni numeriche e una scala di colori (“Blues”) che aiuta a individuare rapidamente dove il modello sbaglia di più.
3. Etichettatura e rendering: assegna i nomi agli assi (“Predicted” e “True”), imposta un titolo (“Confusion”) e infine visualizza il grafico a schermo con **plt.show()**. In questo modo ottieni sia un sommario testuale delle prestazioni sia un’analisi visiva degli errori del tuo classificatore.

Questa è la classification Report:

Test accuracy: 0.9897				
Classification Report:				
	precision	recall	f1-score	support
0	1.00	0.99	0.99	980
1	0.99	1.00	0.99	1135
2	1.00	0.98	0.99	1032
3	0.99	0.99	0.99	1010
4	0.99	0.99	0.99	982
5	0.99	0.99	0.99	892
6	0.99	0.99	0.99	958
7	0.99	0.98	0.99	1028
8	0.97	0.99	0.98	974
9	0.99	0.99	0.99	1009
accuracy			0.99	10000
macro avg	0.99	0.99	0.99	10000
weighted avg	0.99	0.99	0.99	10000

Questa è la matrice di confusione:



Notiamo ad esempio che l'8 è una classe difficile da interpretare, mentre l'1 una delle più facili.

## 7.5 Come includere PyTorch and EDA nel nostro progetto con DevOps

Partiamo da questo script:

```
"""  
Install requirements with :  
pip install torch torchvision scikit-learn matplotlib seaborn  
    tqdm  
"""  
  
import time  
import torch  
import torch.nn as nn  
import torchvision  
import torchvision.transforms as transforms  
from tqdm import tqdm  
import numpy as np  
import matplotlib.pyplot as plt  
import seaborn as sns  
from sklearn.metrics import classification_report ,  
    confusion_matrix  
# Automatically download and load MNIST data  
transform = transforms.Compose([transforms.ToTensor()])  
train_set = torchvision.datasets.MNIST(root = './ data' ,  
    train = True ,  
    download = True , transform = transform )  
test_set = torchvision.datasets.MNIST( root = './ data' ,  
    train = False ,  
    download = True , transform = transform )  
train_loader = torch.utils.data.DataLoader( train_set ,  
    batch_size  
=64 , shuffle = True )
```

```
test_loader = torch . utils . data . DataLoader ( test_set ,  
    batch_size  
=1000 , shuffle = False )
```

Aggiorniamo il requirements.txt:

- Ricordiamoci di aggiungere tutti i nuovi pacchetti visti.
- Potremmo anche pensare di avere 2 diversi requirements, per esempio in produzione, dell' lint, del test e della coverage non ce ne facciamo niente.



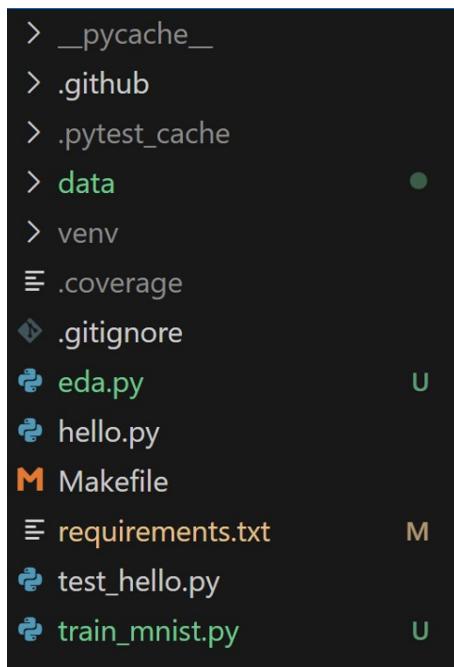
```
☰ requirements.txt  
1  pylint  
2  pytest  
3  pytest-cov  
4  torch  
5  torchvision  
6  scikit-learn  
7  matplotlib  
8  seaborn  
9  tqdm  
10
```

Iniziamo a creare i file:

1. Un file per l'EDA sul nostro dataset.
2. Uno per il training separatamente.

La nostra cartella del progetto sarà strutturata in questo modo

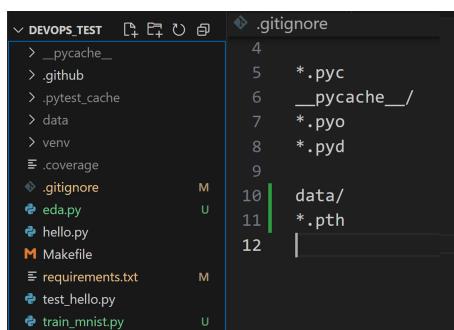
- Un file per l'EDA (Eda.py)
  - Un file per il training (Train\_mnist.py)
  - Se non ignorati i dati verranno aggiunti, ricordiamoci quindi di cancellare nel gitignore, non li vogliamo nel source control.



```
#GITIGNORE FILE (.gitignore)
*.pyc
__pycache__/
*.pyo
*.pyd

data/
*.pth
```

Ci ritroveremo con questa situazione:



Vediamo la cartella data grigia ora quindi non inclusa. Escludiamo anche i file .pth, sono l'estensione dei dati della rete neurale addestrata. Cerchiamo ora di

riorganizzare il tutto:

- creiamo una cartella per i test chiamata **tests**
- creiamo una cartella per gli script chiamata **src**
- lasciamo fuori solo:
  - requirements
  - MakeFile
  - .gitignore

A volte python non è capace di risolversi i path dei vari file, dobbiamo quindi modificare questa cosa:

```
tests > 🐍 test_hello.py > ...
      1   from src.hello import add
      2
>  3   def test_add():
      4       assert add(1, 2) == 3
      5       assert add(-1, 1) == 0
      6       assert add(0, 0) == 0
      7       assert add(-1, -1) == -2
      8       assert add(1000000, 2000000) == 3000000
      9
```

Facciamo ora un po di codice per il train\_mnist :

```
class SimpleCNN(nn.Module):
    def __init__(self):
        super(SimpleCNN, self).__init__()
        self.conv_layers = nn.Sequential(
            nn.Conv2d(1, 32, kernel_size=3),
            nn.ReLU(),
            nn.MaxPool2d(2),
            nn.Conv2d(32, 64, kernel_size=3),
            nn.ReLU(),
            nn.MaxPool2d(2)
        )
        self.fc_layers = nn.Sequential(
            nn.Flatten(),
            nn.Linear(64 * 5 * 5, 64), # output size after conv
            nn.ReLU(),
            nn.Linear(64, 10)
        )

    def forward(self, x):
```

```

        x = self.conv_layers(x)
        x = self.fc_layers(x)
        return x

def train_mnist(epochs: int, save_path: str = "mnist_model.pth"):
    def evaluate_mnist(model_path: str = "mnist_model.pth"):

        if __name__ == "__main__":
            train_mnist(5)
            evaluate_mnist()

```

↑ Questo codice definisce una rete neurale convoluzionale chiamata **SimpleCNN** utilizzando PyTorch, con due blocchi convoluzionali seguiti da due strati completamente connessi. La rete è progettata per classificare immagini in 10 classi, come quelle del dataset MNIST. Sono presenti due funzioni **train\_mnist** e **evaluate\_mnist**, ma non sono implementate. Nella sezione `if __name__ == "__main__"`, viene chiamata **train\_mnist(5)** per addestrare la rete per 5 epoche, e successivamente **evaluate\_mnist()** per valutarla, utilizzando i percorsi di default per salvare e caricare il modello (**mnist\_model.pth**). Tuttavia, il codice non può essere eseguito correttamente finché le due funzioni non vengono implementate.

La parte di training è così strutturata:

```

def train_mnist(epochs: int, save_path: str = "mnist_model.pth"):
    # Automatically download and load MNIST data
    transform = transforms.Compose([transforms.ToTensor()])
    train_set = torchvision.datasets.MNIST(root='./data', train=True,
                                            download=True, transform=transform)

    train_loader = torch.utils.data.DataLoader(train_set,
                                                batch_size=64, shuffle=True)

    model = SimpleCNN()
    criterion = nn.CrossEntropyLoss()
    optimizer = optim.Adam(model.parameters(), lr=0.001)

    # Train the model on CPU
    start = time.time()
    for epoch in tqdm(range(epochs)):
        model.train()
        running_loss = 0.0
        for images, labels in train_loader:
            optimizer.zero_grad()
            outputs = model(images)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()
            running_loss += loss.item()
        print(f'Epoch [{epoch + 1}/5], Loss: {running_loss / len(
            train_loader) :. 4f}')
    print(f'Training time: {time.time() - start :. 2f} seconds')

    torch.save(model.state_dict(), save_path)

```

↑ La funzione train\_mnist addestra il modello SimpleCNN sul dataset MNIST per un numero di epoche specificato. Il dataset viene scaricato automaticamente, trasformato in tensori e caricato in batch da 64. Viene istanziato il modello, definita la funzione di perdita e l'ottimizzatore Adam. Per ogni epoca, il modello viene messo in modalità training e addestrato su tutti i batch. Dopo ogni epoca viene stampata la loss media. Alla fine del training, viene stampato il tempo totale impiegato e salvato lo stato dei pesi del modello nel percorso specificato.

La parte dell'evaluate\_mnist è così:

```
def evaluate_mnist(model_path: str = "mnist_model.pth"):
    # Automatically download and load MNIST data
    transform = transforms.Compose([transforms.ToTensor()])
    test_set = torchvision.datasets.MNIST(root='./data', train=
        False, download=True, transform=transform)
    test_loader = torch.utils.data.DataLoader(test_set, batch_size
        =1000, shuffle=False)

    assert os.path.exists(model_path), f"Model file {model_path}\
        does not exist."

    model = SimpleCNN()
    model.load_state_dict(torch.load(model_path))

    # Evaluate the model
    model.eval()
    correct = 0
    total = 0
    all_preds = []
    all_labels = []

    with torch.no_grad():
        for images, labels in test_loader:
            outputs = model(images)
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()
            all_preds.extend(predicted.numpy())
            all_labels.extend(labels.numpy())

    accuracy = correct / total
    print(f'\nTest accuracy: {accuracy:.4f}')

    # Classification report
    print("\nClassification Report:")
    print(classification_report(all_labels, all_preds))

    # Plot confusion matrix
    conf_matrix = confusion_matrix(all_labels, all_preds)
    plt.figure(figsize=(8,6))
    sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Blues')
    plt.xlabel('Predicted')
    plt.ylabel('True')
    plt.title('Confusion Matrix')
    plt.show(block=False)
```

```
plt.savefig("confusion_matrix.png")
```

↑ Il codice serve a valutare un modello di rete neurale convoluzionale (SimpleCNN) già addestrato sul dataset MNIST, che contiene immagini di cifre scritte a mano. Una volta caricato il modello da file, il codice:

1. Scarica e prepara automaticamente il set di test MNIST, normalizzando le immagini.
2. Esegue la valutazione del modello in modalità "no\_grad" (senza aggiornare i pesi), calcolando il numero di predizioni corrette sul test set.
3. Stampa l'accuratezza del modello sul test set.
4. Genera un classification report contenente precision, recall e F1\_score per ciascuna cifra (da 0 a 9).
5. Crea e salva un'immagine della confusion matrix, che mostra come il modello ha classificato ogni cifra, facilitando l'analisi degli errori.

In sintesi, il codice fornisce una valutazione completa delle performance del modello su dati mai visti prima.

Detto questo, come facciamo a fare Continous Integration di tutto ciò che abbiamo visto? Per testarla dovremo fare il training di tutto, ma non si può fare ad ogni commit questa cosa, dobbiamo addestrare per ogni epoca.

```
import os
from src.train_mnist import train_mnist, evaluate_mnist

def test_mnist_training():
    test_model_path = "test_mnist_model.pth"
    # Train for 1 epoch
    train_mnist(1, save_path=test_model_path)
    # Check if the model file is created
    assert os.path.exists(test_model_path), "Model file not found after training."
    # Check if the model file is not empty, and can be loaded
    assert os.path.getsize(test_model_path) > 0, "Model file is empty."
    # Load the model to ensure it can be loaded without errors
    import torch
    from src.train_mnist import SimpleCNN
    model = SimpleCNN()
    model.load_state_dict(torch.load(test_model_path))

    # Do a simple inference to check if the model is working
    xin = torch.randn(1, 1, 28, 28) # Random input tensor
    model.eval()
    with torch.no_grad():
        output = model(xin)
    assert output.shape == (1, 10), "Model output shape is incorrect."
    # Clean up the model file after the test
```

```

os.remove(test_model_path)
os.remove("confusion_matrix.png")
print("Test passed: Model trained and saved successfully.")

def test_mnist_evaluation():
    # Train the model first
    test_model_path = "test_mnist_model.pth"
    train_mnist(1, save_path=test_model_path)
    evaluate_mnist(test_model_path)
    os.remove(test_model_path)

```

Il codice contiene due funzioni di test che servono per verificare che il processo di training e valutazione di un modello su MNIST funziona correttamente.

1. `test_mnist_training()`: Questa funzione testa che l'addestramento e il salvataggio del modello funzionino come previsto:
  - Allena il modello per una sola epoca e lo salva su file.
  - Verifica che il file del modello esista e che non sia vuoto.
  - Prova a caricare il modello salvato per assicurarsi che sia leggibile.
  - Esegue una inferenza con un input casuale (simulando un'immagine MNIST), per controllare che la forma dell'output sia corretta (vettore di 10 classi, una per cifra da 0 a 9).
  - Elimina i file generati (.pth e immagine della confusion matrix) per pulire l'ambiente di test.
  - Stampa un messaggio di successo.
2. `test_mnist_evaluation()`: Questa funzione serve a testare la valutazione del modello:
  - Addestra nuovamente il modello per una sola epoca.
  - Valuta il modello usando la funzione `evaluate_mnist`, che calcola accuratezza, classification report e confusion matrix.
  - Alla fine, rimuove il file del modello.
3. L'obiettivo complessivo è assicurarsi che:
  - Il modello possa essere addestrato, salvato e ricaricato senza errori.
  - L'inferenza funzioni come previsto.
  - Il processo di valutazione non generi eccezioni o comportamenti anomali.

L'EDA (codice visto primo di creazione dei plot ...), è stato messo dentro un file chiamato eda, dentro una cartella chiamata eda. Quando faccio l'eda semplicemente lo eseguo, vedo che mi vengano creati i file e poi li cancello. Se tutto funziona dovrebbe essere apposto.

```

import os
from src.eda import eda

def test_ed():
    ed() # Run the EDA function to check if it executes without
        # errors
    os.remove("class_distribution.png")
    os.remove("sample_images.png")
    print("EDA test passed: EDA function executed successfully.")

```

Andiamo ad aggiornare anche il MakeFile, rimane praticamente uguale:

```

M Makefile
1 install:
2     python -m pip install --upgrade pip && pip install -r requirements.txt
3     @echo "Installation complete. You can now run the project."
4
5 lint:
6     pylint --disable=R,C src/*.py tests/*.py
7     @echo "Linting complete. No issues found."
8
9 test:
10    python -m pytest -vv --cov=src tests/
11    @echo "Testing complete. All tests passed."
12

```

Alla fine eseguo, faccio `make test` ed ottengo questi risultati:

```

(venv) (base) PS D:\Github\devops_test> make test
python -m pytest -vv --cov=src tests/
=====
platform win32 -- Python 3.12.8, pytest-8.3.5, pluggy-1.5.0 -- D:\Github\devops_test\venv\Scripts\python.exe
cachedir: .pytest_cache
rootdir: D:\Github\devops_test
plugins: cov-6.1.1
collected 4 items

tests/test_ed.py::test_ed PASSED
tests/test_hello.py::test_add PASSED
tests/test_train.py::test_mnist_training PASSED
tests/test_train.py::test_mnist_evaluation PASSED

=====
          tests coverage -----
coverage: platform win32, python 3.12.8-final-0

Name      Stmts  Miss  Cover
-----
src\eda.py      39      2   95%
src\hello.py     3      0  100%
src\train_mnist.py  76      2   97%
TOTAL       118      4   97%
=====
4 passed in 29.33s
=====
Testing complete. All tests passed.

```

Una cosa importante è la coverage, ovvero il controllo che tutte le righe del mio script siano state eseguite.

## 7.6 Pytorch Lightning

È una libreria che ottimizzano il lavoro di addestramento delle reti neurali, ad esempio, nel codice che vediamo ↓, gli diamo il modello in pasto e sarà lui che si occuperà di fare il training del modello.

```
import torch
import lightning as L

class ToyExample(L.LightningModule):
    def __init__(self, model):
        super().__init__()
        self.model = model

    def training_step(self, batch):
        loss = self.model(batch).sum()
        return loss

    def configure_optimizers(self):
        return torch.optim.Adam(self.model.parameters())

if __name__ == "__main__":
    model = torch.nn.Linear(32, 2)
    pl_module = ToyExample(model)
    train_dataloader = torch.utils.data.DataLoader(torch.randn(8, 32))
    trainer = L.Trainer()
    trainer.fit(pl_module, train_dataloader)
```

Questo codice implementa un esempio minimale di training utilizzando il framework PyTorch Lightning, con lo scopo di dimostrare la struttura base di un modello senza un obiettivo pratico specifico. Ecco cosa fa nel complesso:

1. Definizione del modello:

- Viene creato un modello di machine learning semplice: una rete neurale con un singolo strato lineare (`nn.Linear`)
- Il modello prende in input 32 valori e restituisce 2 valori in output.

2. Setup del training:

- Dati casuali: I dati di training sono generati casualmente (tensori di forma 8x32), simulando un dataset con 8 esempi e 32 feature ciascuno.
- Ottimizzatore: Viene configurato l'ottimizzatore Adam per aggiornare i parametri del modello.

3. Logica di training:

- Il modello elabora un batch di dati.
- La loss è calcolata come somma di tutti i valori in output del modello (invece di una loss tradizionale come MSE o CrossEntropy).
- Esempio: se l'output è  $[[0.1, -0.2], [0.3, 0.4]]$ , la loss sarà  $0.1 - 0.2 + 0.3 + 0.4 = 0.6$ .

- Obiettivo: Minimizzare questa loss arbitraria (senza un significato statistico reale).

4. Esecuzione:

- Il codice allena il modello per 1 epoch (poiché il dataloader ha un solo batch di 8 esempi).
- Usa il Trainer di PyTorch Lightning, che gestisce automaticamente:
  - L'aggiornamento dei parametri.
  - La distribuzione su hardware (CPU/GPU se disponibile).
  - Il logging (non presente in questo esempio).

## 7.7 Come gestiamo i modelli che già esistono?

Esistono diversi siti in cui ci sono modelli già addestrati, che ci danno accesso a modelli con dataset o approcci già settati e preaddestrati. Andrò quindi, in base alle mie esigenze, a scegliere i modelli dai vari gruppi di ricerca e dalle varie repository. Per il progetto si consiglia il sito Hugging Face.

## 8 iperparametri

Sono una serie di parametri che non controllano di per sé il comportamento del modello, bensì ne controllano la capacità e le performance durante il training. Abbiamo ad esempio:

- Learning rate: quanto rapidamente il modello deve imparare, più è alto più inizialmente imparerà in fretta, se è troppo alto dopo un tot non riuscirà a diventare più bravo.
- Dropout: valori casuali che vengono fatti durante le operazioni di training.
- Tipo di ottimizzatore.
- Weight decay.
- Activation functions.
- Scheduler parameters.

Tutti questi iperparametri, sono dei processi che richiedono un tuning manuale ed è molto time-consuming. Ci sono dei parametri che con ricerca a tappeto (grid search) puoi andare ad esplorare e vedere quali sono quelli ottimi per il mio sistema.

## 8.1 Grid Search: Idea

- Decido una griglia di iperparametri.
- Facciamo training e testing di tutti questi valori.
- Seleziono quelli che funzionano meglio.

Per fare un buon lavoro dobbiamo quindi decidere il mio modello, decidere quali sono gli iperparametri da andare ad ottimizzare, facciamo dei cicli di loop iterativi per lavorare su tutte le possibili combinazioni, li associo al mio training set e avrò i migliori parametri. Con strumenti come Optuna si può fare tutto questo lavoro in automatico, farà un buon sampling dei vari parametri tramite suoi algoritmi interni.

```
# optuna_search.py
import optuna
import torch
from torchvision import datasets, transforms

def objective(trial):
    # Hyperparameters to tune
    lr = trial.suggest_loguniform('lr', 1e-5, 1e-1)
    dropout = trial.suggest_float('dropout', 0.1, 0.5)

    # Model definition
    model = torch.nn.Sequential(
        torch.nn.Flatten(),
        torch.nn.Linear(28*28, 128),
        torch.nn.ReLU(),
        torch.nn.Dropout(dropout),
        torch.nn.Linear(128, 10)
    )

    # Training loop (one epoch for demo)
    optimizer = torch.optim.Adam(model.parameters(), lr=lr)
    criterion = torch.nn.CrossEntropyLoss()
    train_loader = torch.utils.data.DataLoader(
        datasets.MNIST('..', train=True, download=True, transform=
            transforms.ToTensor()),
        batch_size=64, shuffle=True
    )

    model.train()
    for batch in train_loader:
        data, target = batch
        optimizer.zero_grad()
        output = model(data)
        loss = criterion(output, target)
        loss.backward()
        optimizer.step()

    # Validation accuracy as objective
    val_loader = torch.utils.data.DataLoader(
        datasets.MNIST('..', train=False, download=True, transform=
            transforms.ToTensor()),
```

```

        batch_size=1000, shuffle=False
    )
    correct = 0
    with torch.no_grad():
        for data, target in val_loader:
            preds = model(data).argmax(dim=1)
            correct += (preds == target).sum().item()
    accuracy = correct / len(val_loader.dataset)
    return accuracy

study = optuna.create_study(direction='maximize')
study.optimize(objective, n_trials=20)

```

↑ Questo codice carica un modello già addestrato per il riconoscimento delle cifre scritte a mano (MNIST), lo valuta sul set di test calcolandone l'accuratezza, stampa un report con le metriche di classificazione e genera una confusion matrix per visualizzare gli errori di predizione.

La cosa principale è decidere quindi quali sono i parametri da addestrare, il modello può essere qualsiasi cosa, i dati possono essere i nostri, facciamo il training e vogliamo massimizzare l'accuratezza.

```

# optuna_search.py
import optuna
import torch
from torchvision import datasets, transforms

def objective(trial):
    # Hyperparameters to tune
    lr = trial.suggest_loguniform('lr', 1e-5, 1e-1)
    dropout = trial.suggest_float('dropout', 0.1, 0.5)

    # Model definition
    model = torch.nn.Sequential(
        torch.nn.Flatten(),
        torch.nn.Linear(28*28, 128),
        torch.nn.ReLU(),
        torch.nn.Dropout(dropout),
        torch.nn.Linear(128, 10)
    )

    # Training loop (one epoch for demo)
    optimizer = torch.optim.Adam(model.parameters(), lr=lr)
    criterion = torch.nn.CrossEntropyLoss()
    train_loader = torch.utils.data.DataLoader(
        datasets.MNIST('.', train=True, download=True, transform=
            transforms.ToTensor()),
        batch_size=64, shuffle=True
    )

    model.train()
    for batch in train_loader:
        data, target = batch
        optimizer.zero_grad()
        output = model(data)
        loss = criterion(output, target)

```

```

        loss.backward()
        optimizer.step()

    # Validation accuracy as objective
    val_loader = torch.utils.data.DataLoader(
        datasets.MNIST('..', train=False, download=True, transform=
            transforms.ToTensor()),
        batch_size=1000, shuffle=False
    )
    correct = 0
    with torch.no_grad():
        for data, target in val_loader:
            preds = model(data).argmax(dim=1)
            correct += (preds == target).sum().item()
    accuracy = correct / len(val_loader.dataset)
    return accuracy

study = optuna.create_study(direction='maximize')
study.optimize(objective, n_trials=20)

```

Ottenendo questi risultati:

```

[I] 2025-05-16 17:03:44,946 A new study created in memory with name: no-name-ef24a1aa-16fe-4c6d-8025-06f15fa3502a
D:\GitHub\devops\devops\test\optuna_search.py:7: FutureWarning: suggest_loguniform has been deprecated in v3.0.0. This feature will be removed in v6.0.0. See https://github.com/optuna/optuna
/releases/tag/v3.0.0. Use suggest_float(..., log=True) instead.
    lr = trial.suggest_loguniform("lr", 1e-5, 1e-1)
100% | 9.91M/9.91M [00:44<00:00, 225kB/s]
100% | 28.9k/28.9k [00:00<00:00, 240kB/s]
100% | 1.65M/1.65M [00:00<00:00, 1.69MB/s]
100% | 4.54k/4.54k [00:00<00:00, 4.55MB/s]
[T] 2025-05-16 17:04:37,427 Trial 0 finished with value: 0.8708 and parameters: {'lr': 0.033650875853008876, 'dropout': 0.2271561761606986}. Best is trial 0 with value: 0.8708.
[T] 2025-05-16 17:04:41,427 Trial 1 finished with value: 0.8351 and parameters: {'lr': 4.6214444042619746e-05, 'dropout': 0.25201593161258734}. Best is trial 0 with value: 0.8708.
[T] 2025-05-16 17:04:45,466 Trial 2 finished with value: 0.8466 and parameters: {'lr': 0.03509057337085886, 'dropout': 0.3114584001385457}. Best is trial 0 with value: 0.8708.
[T] 2025-05-16 17:04:49,546 Trial 3 finished with value: 0.8467 and parameters: {'lr': 4.866346226378741e-05, 'dropout': 0.213802940835008538}. Best is trial 0 with value: 0.8708.
[T] 2025-05-16 17:04:54,574 Trial 4 finished with value: 0.9318 and parameters: {'lr': 0.007380622633304385, 'dropout': 0.2912752448812965}. Best is trial 4 with value: 0.9318.
[T] 2025-05-16 17:05:00,827 Trial 5 finished with value: 0.8968 and parameters: {'lr': 0.0015806387562415953, 'dropout': 0.17579701139941872}. Best is trial 4 with value: 0.9318.
[T] 2025-05-16 17:05:07,767 Trial 6 finished with value: 0.8608 and parameters: {'lr': 0.04007528161806580, 'dropout': 0.19171533066891913}. Best is trial 4 with value: 0.9318.
[T] 2025-05-16 17:05:14,942 Trial 7 finished with value: 0.6533 and parameters: {'lr': 0.07881404831036119, 'dropout': 0.24928787662990817}. Best is trial 4 with value: 0.9318.
[T] 2025-05-16 17:05:22,722 Trial 8 finished with value: 0.8454 and parameters: {'lr': 0.043249256935530535, 'dropout': 0.20275594854729753}. Best is trial 4 with value: 0.9318.
[T] 2025-05-16 17:05:29,388 Trial 9 finished with value: 0.937 and parameters: {'lr': 0.010705772762708678, 'dropout': 0.17814453035512906}. Best is trial 9 with value: 0.937.
[T] 2025-05-16 17:05:36,499 Trial 10 finished with value: 0.9344 and parameters: {'lr': 0.0020350708176499624, 'dropout': 0.43797803461254564}. Best is trial 9 with value: 0.937.
[T] 2025-05-16 17:05:41,737 Trial 11 finished with value: 0.933 and parameters: {'lr': 0.001854069698630163931, 'dropout': 0.45289453730553495}. Best is trial 9 with value: 0.937.
[T] 2025-05-16 17:05:46,012 Trial 12 finished with value: 0.9272 and parameters: {'lr': 0.0012808230881383121, 'dropout': 0.4952581943882971}. Best is trial 9 with value: 0.937.
[T] 2025-05-16 17:05:49,962 Trial 13 finished with value: 0.9436 and parameters: {'lr': 0.0046430914688045, 'dropout': 0.38098619950009155}. Best is trial 13 with value: 0.9436.
[T] 2025-05-16 17:05:53,676 Trial 14 finished with value: 0.9549 and parameters: {'lr': 0.006676519823140869, 'dropout': 0.108071748801502306}. Best is trial 14 with value: 0.9549.
[T] 2025-05-16 17:05:57,427 Trial 15 finished with value: 0.91 and parameters: {'lr': 0.0002749821731610396, 'dropout': 0.1099012774283893}. Best is trial 14 with value: 0.9549.
[T] 2025-05-16 17:06:01,701 Trial 16 finished with value: 0.9186 and parameters: {'lr': 0.0080900022183064925, 'dropout': 0.3590433299528462}. Best is trial 14 with value: 0.9549.
[T] 2025-05-16 17:06:04,755 Trial 17 finished with value: 0.9011 and parameters: {'lr': 0.0002959261957087064, 'dropout': 0.3722114674435642}. Best is trial 14 with value: 0.9549.
[T] 2025-05-16 17:06:08,520 Trial 18 finished with value: 0.7673 and parameters: {'lr': 1.1042088528082772e-05, 'dropout': 0.10493021746457447}. Best is trial 14 with value: 0.9549.
[T] 2025-05-16 17:06:13,370 Trial 19 finished with value: 0.9302 and parameters: {'lr': 0.004353991874558065, 'dropout': 0.3624183977601863}. Best is trial 14 with value: 0.9549.
Best trial: {'lr': 0.006676519823140869, 'dropout': 0.10671748801502306}

```

Alla fine ci dirà che il migliore è quello che ha il Learning Rate di 0.00667... e il dropout di 0.1 Questo vale di più per la parte sperimentale.

## 9 Docker, Logging e Monitoring

Docker è una tecnologia che permette di rilasciare software su delle macchine, andando a standardizzare l'installazione e l'esecuzione del codice. Togliamo così tutti i problemi relativi all'utilizzo del codice su macchine diverse dalla nostra. Lato codice ci abbiamo già pensato un po' con DevOps. Con il Docker andremo quindi a standardizzare la parte di esecuzione. Guarderemo anche un po' il monitoring ed il logging, non li vedremo però applicati.

### 9.1 Docker

Come detto prima abbiamo dei problemi da risolvere sul nostro software:

- Identifico questo problema.
- Scrivo il codice.
- Lo testo sulla mia macchina.
- Infine lo metto in produzione. Qui troveremo probabilmente degli errori quindi:
  - Controlleremo che il nostro codice sia apposto.
  - Lo verifichiamo localmente.
  - Faccio Deploy.
  - E via dicendo.

Con molta probabilità ci ritroveremo comunque con degli errori su altre macchine.

Entra quindi in gioco Docker, ma cosa fa?

- Crea i **Containers**, sono sostanzialmente delle soluzioni leggere, portatili ed autocontenute. In pratica definiamo le nostre specifiche una sola volta e siamo capaci di eseguire il nostro codice ovunque.
- Docker non risolve completamente il problema, va installato su tutte le macchine, una volta che tutti hanno l'installazione ed è standardizzata possiamo lavorare su più sistemi per l'appunto.

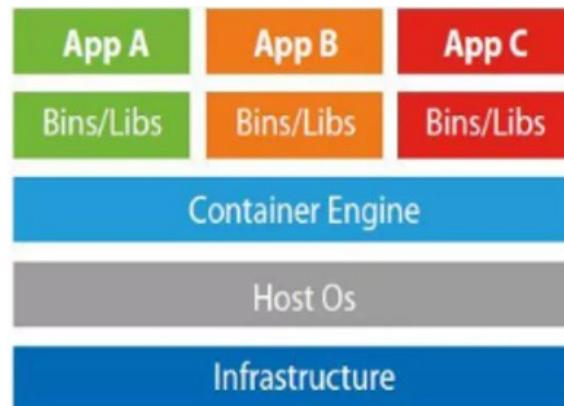
## 9.2 Containerization

È una sorta di versione evoluta della virtualizzazione, non è virtualizzazione vera e propria, il vantaggio è che è facile replicarlo. È modulare, posso quindi avere più container, anche grandi, che comunicano tra di loro, garantendoci grande scalabilità. È abbastanza rapido, ha un overhead abbastanza basso non avendo tutto il sistema operativo sotto, d'altra parte è vero che per le applicazioni di Machine Learning non è velocissimo. Questo succede perché quando vado a toccare l'hardware, non è più solo docker che lavora ma anche la macchina. È molto efficiente perché ci permette di separare bene i microservizi, ed è portatile, basta solo la configurazione (potendolo così personalizzare). I concetti base sono dunque:

- Container: pacchetto autocontenuto...
- Immagini: il nostro software crea una cosiddetta immagine ed un'istanza dell'immagine è un Container. Quindi se io ho un'immagine che rappresenta il mio programma e lancio 50 volte il mio programma, sto lanciando 50 container della mia immagine.
- Non c'è un sistema di virtualizzazione, quindi non c'è un overhead al lancio dell'applicativo, avremo quindi delle performance migliori.
- Gestisce da solo le risorse.

### 9.2.1 Docker vs Virtual Machine

I container non sono Virtual Machine, sostanzialmente invece che avere il OS, i driver, i kernel... Nella VM avremo quindi un sacco di lavoro tra macchina e OS. Docker è un processo isolato, ha bisogno soltanto dei file per l'esecuzione. Possiamo mandare in esecuzione diversi container, ma condividono tutte lo stesso kernel, potendo avere un impatto infrastrutturale ridotto. Questa è una rappresentazione visiva di come è strutturato ↓



Questa invece è la rappresentazione con VM ↓



Vediamo quindi la differenza, il OS in più è quello che appesantirà il lavoro, insieme agli Hypervisor. I benefici sono soprattutto:

- Veloce da sviluppare, una volta che abbiamo l'applicazione fatta bastano poche righe di codice per farlo.

- Veloce da rilasciare, lanciare il container è una riga.
- Puoi fare molto bene il controllo sui servizi.
- È molto comodo grazie all'autorelease.

Le piattaforme possiamo usare sia windows che Linux(originariamente è stato progettato per Linux), ora con WSL possiamo girarlo nativamente su windows in maniera più diretta. C'è Docker Desktop, non è perfetto proprio perchè non gira su un sistema operativo Linux, non va molto bene per addestramento con GPU.

### 9.2.2 Docker commands

I comandi (su terminale) generali di Docker sono:

- docker <main command><secondary command> [parameters]
- scrivendo docker vedremo i comandi disponibili
- per la versione basterà scrivere docker version o docker -v

## 9.3 Immagine

Sono una serie di file che vengono cumulativamente messi uno sull'altro formando un punto di partenza del nostro codice, in pratica prendo il codice da zero, installo tutti i requirements, fatto questo diventa un'immagine. Generalmente si prende un'immagine pre-costruita per poi personalizzarla. Ogni cambiamento a queste immagini diventa un layer, questo layer ha un identificativo univoco e viene salvato sulla macchina dove stiamo compilando Docker. In pratica dal nostro punto di partenza, se vogliamo installare i nostri requisiti lo diciamo a Docker e lui prenderà l'immagine originale, che diventerà un layer di partenza, creando alla fine un'immagine composta dal layer di partenza e quello che abbiamo modificato ottenendo così un'immagine. Ogni layer pesa, quindi bisogna cercare di creare meno layer possibili.

Alcuni comandi sono:

- Per vedere le immagini in cache locale: docker image ls
- Per scaricare un'immagine dal docker hub: docker pull <nomeImmagine>
- Per cancellare un'immagine: docker image rm <nomeImmagine>
- Per rimuovere le immagini non usate: docker image prune

Dove vengono mandate le immagini? Le immagini possono essere salvate in locale, se voglio però distribuirli su altri computer abbiamo Docker Hub. È un contenitore condiviso in cui possiamo condividere le nostre immagini e prenderne altre già fatte. Da qui ci basterà fare `docker pull hello-world`, e lui ci scaricherà l'immagine, se non la trova lui ci scaricherà l'ultima tramite il tag latest. La pulla da library/Hello-World, una volta scaricata per caricare l'immagine ci basterà scrivere `docker run nome-immagine`. Ogni volta che eseguiamo un container gli diamo

un nome temporaneo, viene eseguito il codice ed una volta eseguito il container muore.

Abbiamo quindi altri comandi utili:

- Per assegnare un taggare un’immagine: docker image tag <nomeImmagine>[:TAG] <nomeImmagine>[:TAG]
- Per vedere la storia di un’immagine: docker image history <nomeImmagine>
- Per vedere il contenuto/configurazione di un’immagine: docker image inspect <nomeImmagine>

Il container come dicevamo è l’istanza di un’immagine ed inizialmente non è altro che una copia, quello che succede è che l’immagine pian piano evolve, ad esempio vengono creati dei file temporanei, viene salvato qualcosa sul disco... Fa dei cambiamenti nel container ma non nell’immagine, infatti se facciamo modifiche al container durante l’esecuzione, quando muore il container perdiamo tutti i file. Non si salva quindi mai nulla nel container, a meno che non sappiamo come recuperarlo, ad esempio tramite i **Volumi**.

Comandi un po’ più operativi:

- Per vedere i container attivi: docker ps
- Per vedere tutti quelli sulla macchina: docker ps -a

### 9.3.1 Docker Run

docker run [-v,-d,-p.options]<nomeImmagine>

- -p <portaHost>:<portaContainer> = espone una porta → Ogni container viene eseguito e comunica su uno stato di rete isolato, per farlo parlare con l’esterno si lancia questo comando. Questo comando espone una porta di Docker con una porta dell’interfaccia di rete della macchina su cui stiamo eseguendo Docker (è un ponte).
- -d = detached mode (resta attivo in background) → Fa semplicemente partire il processo in background lasciando al sistema operativo la gestione.
- -v <nomeVolume>:<percorsoNelContainer> = Assegna un volume definito → È il percorso assoluto nella macchina, contro il percorso che avrà poi nel container (lo vedremo tra poco).
- -rm =non lascia traccia del container al termine → È utile se vogliamo eliminare l’istanza del container, utile quando il container deve nascere e morire senza popolare la lista.
- docker run -it <nomeImmagine> bash → per avviare un’immagine ed entrare nel suo terminale, possiamo entrare in quel container come se fossimo all’interno di quella macchina (utile per il debug).

- docker container exec -it <nomeContainer> <bash|sh> → se vogliamo entrare nella shell di un container già in esecuzione.
- docker container port <nomeContainer> → per vedere le porte condivise tra host e container.

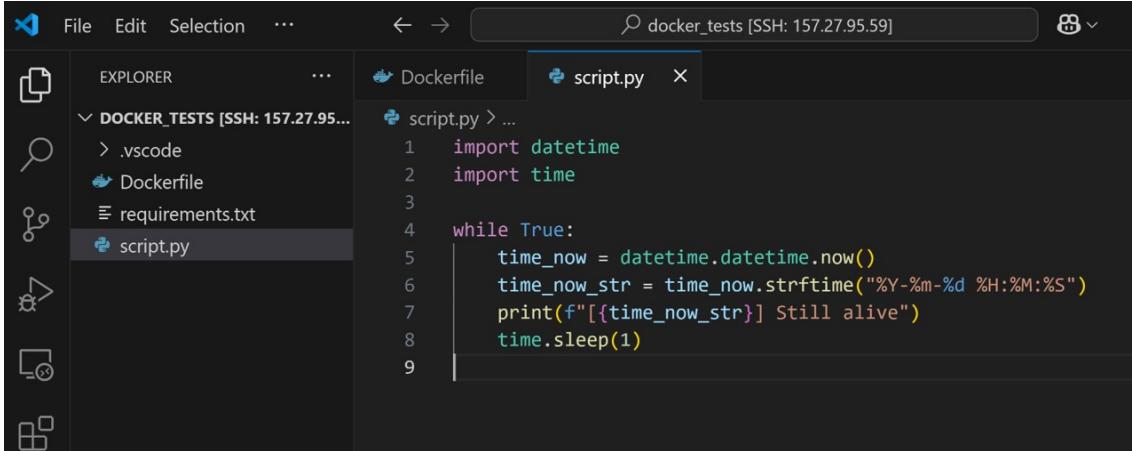
### 9.3.2 DOckerfile

È un semplice file di testo senza estensione (la si aggiunge solo se vogliamo fare versioni diverse). Ci serve un’immagine da cui partire ed ogni nuova riga che aggiungiamo è un nuovo layer, l’insieme di tutti quanti i layer è un’immagine. Più layer utilizziamo più spazio utilizziamo, quindi più modifiche verranno fatte durante la nostra build.

Comandi importanti del dockerfile:

- **ADD:** ci permettere di aggiungere dati direttamente dal sistema operativo corrente al container.
- **ARG:** se vogliamo specificare delle variabili che vengono messe durante la fase di compilazione.
- **CMD:** specifica il comando da eseguire alla fine dell’esecuzione del container.
- **COPY:** come l’add ma qui puoi mantenere la struttura delle cartelle.
- **ENTRYPOINT:** ci dice qual’è l’ eseguibile di partenza del nostro container.
- **ENV:** per descrivere delle variabili d’ambiente.
- **EXPOSE:** per dire quali porte sono aperte.
- **FROM:** crea una nuova build per un’immagine di partenza.
- **HEALTHCHECK:** per fare un check prima della partenza del container.
- **LABEL:** se vogliamo aggiungere dei metadati.
- **MAINTAINER:** specifica l’autore dell’immagine.
- **RUN:** ogni run crea un nuovo layer.
- **WORKDIR:** ci permette di dire in quale cartella tutte le operazioni devono essere eseguite.

Ecco un esempio:

A screenshot of the Visual Studio Code interface. The title bar says "docker\_tests [SSH: 157.27.95.59]". The left sidebar shows a file tree with a folder named "DOCKER\_TESTS [SSH: 157.27.95.59]" containing ".vscode", "Dockerfile", "requirements.txt", and "script.py". The "script.py" file is selected and open in the main editor area. The code in "script.py" is:

```
import datetime
import time

while True:
    time_now = datetime.datetime.now()
    time_now_str = time_now.strftime("%Y-%m-%d %H:%M:%S")
    print(f"[{time_now_str}] Still alive")
    time.sleep(1)
```

Qui è stato creato uno script con un ciclo infinito, prende il tempo corrente e lo formatta in ore, minuti, secondi e anno, mese, giorno; ogni secondo stampa. Ed infine il DockerFile (ogni riga crea un nuovo layer):

```
from python:3.11-slim

#set the working directory
WORKDIR /app #viene creata automaticamente una cartella se non esiste
#copy the requirements file into the container
COPY requirements.txt #copio il file in . ovvero la cartella corrente
#install the dependencies
RUN pip install --no-cache-dir -r requirements.txt #istruzione che mi permette di eseguire un comando come fosse una cmd normale
#copy the rest of the application code into the container
COPY . . #copia tutti file che ci sono nella cartella all'interno della cartella di destinazione

#run python script at container startup
#-u flag is used to force the stdout and stderr streams to be unbuffered, which can be useful for real-time logging
CMD ["python", "-u", "script.py"]
```

↑ Questo Dockerfile crea un'immagine Docker per eseguire uno script Python, seguendo una struttura ottimizzata:

1. Base Image: Parte da un'immagine ufficiale leggera di Python 3.11 (python:3.11-slim), ideale per applicazioni minimaliste.
2. Configurazione ambiente: Imposta la directory di lavoro /app (creata automaticamente se non esiste).
3. Gestione dipendenze:
  - Copia solo il file requirements.txt nella directory di lavoro

- Installa le librerie Python specificate senza memorizzare cache, riducendo le dimensioni finali dell'immagine.
  4. Deploy codice: Dopo l'installazione delle dipendenze, copia tutto il codice sorgente rimanente (file e cartelle locali) nella directory /app.
  5. Esecuzione automatica: All'avvio del container, esegue automaticamente: `python -u script.py`. Il flag `-u` garantisce un output immediato dei log (senza buffering), cruciale per monitoraggio in tempo reale.

In pratica costruisce un ambiente containerizzato pronto a eseguire lo script `script.py` con tutte le sue dipendenze, ottimizzato per dimensioni ridotte e logging efficiente. Per fare il building devo scrivere:

```
docker build -t<imageName> o docker build -t<imageName> -f./Dockerfile
```

```
(base) ffuncuno@frigobar:~/docker_tests$ docker build -t test .
[+] Building 3.4s (10/10) FINISHED                                            docker:default
=> [internal] load build definition from Dockerfile                         0.0s
=> => transferring dockerfile: 388B                                         0.0s
=> [internal] load metadata for docker.io/library/python:3.11-slim          0.4s
=> [internal] load .dockerrcignore                                         0.0s
=> => transferring context: 2B                                           0.0s
=> [1/5] FROM docker.io/library/python:3.11-slim@sha256:9c85d1d49df54abca1c5db3b4016400e198 1.0s
=> => resolve docker.io/library/python:3.11-slim@sha256:9c85d1d49df54abca1c5db3b4016400e198 0.0s
=> => sha256:457229a5b8524ec116a3d28cf70e3af8d5784d5ef5a34d48b0f5109b875 16.21MB / 16.21MB 0.5s
=> => sha256:b658f584ba6ea8b3aaa0393decd85fea571114b5e3a214265e6fd238640e2153 249B / 249B 0.4s
=> => sha256:9c85d1d49df54abca1c5db3b4016400e198e9b9b699f32f1ef8e5c0c2149c 9.13kB / 9.13kB 0.0s
=> => sha256:7da11aeb3f33c4ab675410f7954c6d1c4ae1212210818698e5dc53279723bf 1.75kB / 1.75kB 0.0s
=> => sha256:bea8499a31bb2a2bb828aa4979cba7450ef6ffc275c184c29a52eb421a57d1 5.37kB / 5.37kB 0.0s
=> => sha256:eb0baa05daea15f7e8602d66b4b8c12ea0f02e8f87a047217697cb9831db35 3.51MB / 3.51MB 0.4s
=> => extracting sha256:eb0baa05daea15f7e8602d66b4b8c12ea0f02e8f87a047217697cb9831db351b 0.1s
=> => extracting sha256:457229a5b8524ec116a3d28cf70e3af8d5784d5ef5a34d48b0f5109b8755d8f 0.5s
=> => extracting sha256:b658f584ba6ea8b3aaa0393decd85fea571114b5e3a214265e6fd238640e2153 0.0s
=> [internal] load build context                                              0.0s
=> => transferring context: 164B                                         0.0s
=> [2/5] WORKDIR /app                                                       0.0s
=> [3/5] COPY requirements.txt .                                             0.0s
=> [4/5] RUN pip install --no-cache-dir -r requirements.txt                1.8s
=> [5/5] COPY . . .                                                       0.0s
=> exporting to image                                                       0.1s
=> => exporting layers                                                     0.1s
=> => writing image sha256:76148e09191eeb296658fc4713635a5c4a23cd3c68f486116a1e04554de007e2 0.0s
=> => naming to docker.io/library/test                                      0.0s
(base) ffuncuno@frigobar:~/docker_tests$
```

Fatto ciò facciamo un test con `docker run test`.

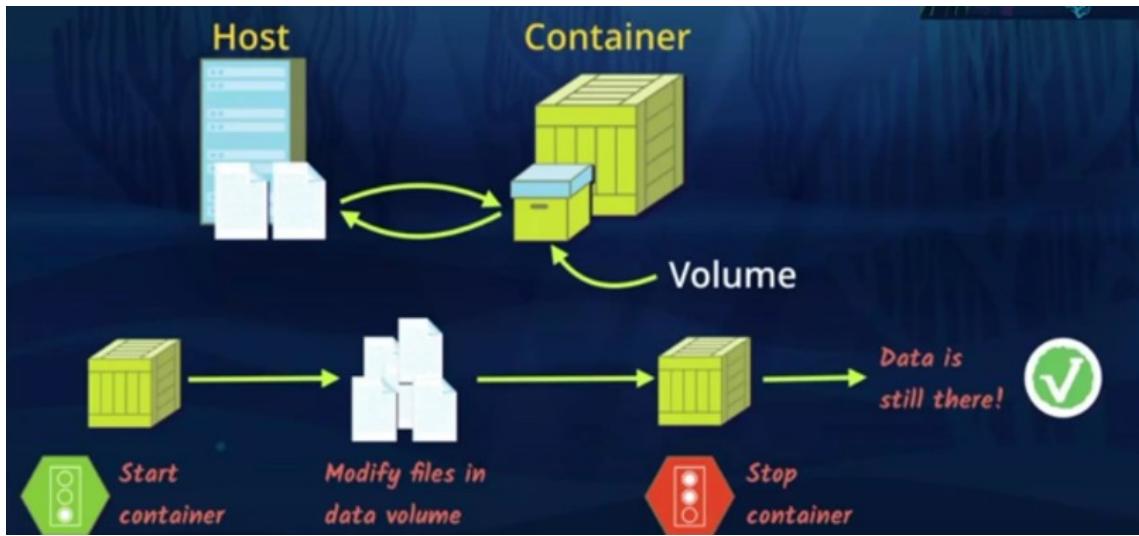
Se voglio lanciarlo in background posso fare `docker run -d test`.

```
(base) fcunico@frigobar:~/docker_tests$ docker run test
[2025-05-21 09:14:55] Still alive
[2025-05-21 09:14:56] Still alive
[2025-05-21 09:14:57] Still alive
[2025-05-21 09:14:58] Still alive
[2025-05-21 09:14:59] Still alive
[2025-05-21 09:15:00] Still alive
[2025-05-21 09:15:01] Still alive
[2025-05-21 09:15:02] Still alive
[2025-05-21 09:15:03] Still alive
[2025-05-21 09:15:04] Still alive
[2025-05-21 09:15:05] Still alive
^CTraceback (most recent call last):
  File "/app/script.py", line 8, in <module>
    time.sleep(1)
KeyboardInterrupt
(base) fcunico@frigobar:~/docker_tests$
```

Il comando docker log test serve per vedere cosa sta loggando, naturalmente qui non troverà nulla, perchè quando ho lanciato docker run gli abbiamo assegnato un nome, quindi dobbiamo andare a cercare quel nome. Con `docker ps` troveremo quello che cerchiamo, in questo caso lucid\_panini. Vedremo poi le informazioni. Facendo `docker log -follow "nome"`, seguirà il trace dello standard output del container fino ad un nostro interrupt da tastiera (ctrl+c). Facendo invece `docker stop "nome"`, fermeremo il container. Possiamo dare un nome al docker con `docer run -d -name "nome" test`.

## 9.4 Volumi

I volumi sono molto importanti perchè ci permettono di passare i dati da dentro a fuori i container e viceversa. Abbiamo essenzialmente una sorta di cartella condivisa, in cui noi facciamo partire il container, modifichiamo dei file all'interno del volume, fermiamo il container e abbiamo ancora i nostri dati in mano.



Come si fa questa cosa ?

Facciamo un esempio:

- Cambio lo script chiamato `create_file_with_content.py`.

```
❸ create_file_with_content.py > ...
1  with open("example.txt", "w") as file:
2      file.write("Hello, this is a test file.")
3      file.write("\nThis file is created using Python.")
4      file.write("\nIt contains multiple lines of text.")
5      file.write("\nThis is the fourth line.")
6      file.write("\nAnd this is the fifth line.")
7      file.write("\nThat's it!")
```

- Apro un file .txt e ci scrivo sopra del contenuto.
- Cambio poi il comando perchè avendo cambiato lo script voglio fare andare il nuovo.

```
CMD ["python", "-u", "create_file_with_content.py"]
```

- Buildo il dockerfile.

```
(base) funico@frigobar:~/docker_tests$ docker build -t test .
[+] Building 2.7s (10/10) FINISHED
=> [internal] load build definition from Dockerfile
=> => transferring dockerfile: 524B
=> [internal] load metadata for docker.io/library/python:3.11-slim
=> [internal] load .dockerignore
=> => transferring context: 2B
=> [1/5] FROM docker.io/library/python:3.11-slim@sha256:9c85d1d49df54abca1c5db3b4016400e198e9e9bb699f32f1ef8e5c0c2149ccf
=> [internal] load build context
=> => transferring context: 1.84kB
=> CACHED [2/5] WORKDIR /app
=> [3/5] COPY requirements.txt .
=> [4/5] RUN pip install --no-cache-dir -r requirements.txt
=> [5/5] COPY . .
=> => exporting to image
=> => exporting layers
=> => writing image sha256:b44c4a736b46138e16a6a10d7b4c5a423f2e02737f3808d65a899db950c2fdda
=> => naming to docker.io/library/test
(base) funico@frigobar:~/docker_tests$
```

Quando arrivo in fondo vado a prendere il mio docker file modificato:

```
Dockerfile X create_file_with_content.py
Dockerfile > ...
1  FROM python:3.11-slim
2
3  # Set the working directory
4  WORKDIR /app
5  # Copy the requirements file into the container
6  COPY requirements.txt .
7  # Install the dependencies
8  RUN pip install --no-cache-dir -r requirements.txt
9  # Copy the rest of the application code into the container
10 COPY . .
11
12 # Run python script at container startup
13 # -u flag is used to force the stdout and stderr streams to be unbuffered
14 # CMD ["python", "-u", "small_yolo_train.py"]
15 CMD ["python", "-u", "create_file_with_content.py"]
16
```

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
(base) funico@frigobar:~/docker_tests$ docker run -v `pwd`:/app test
```

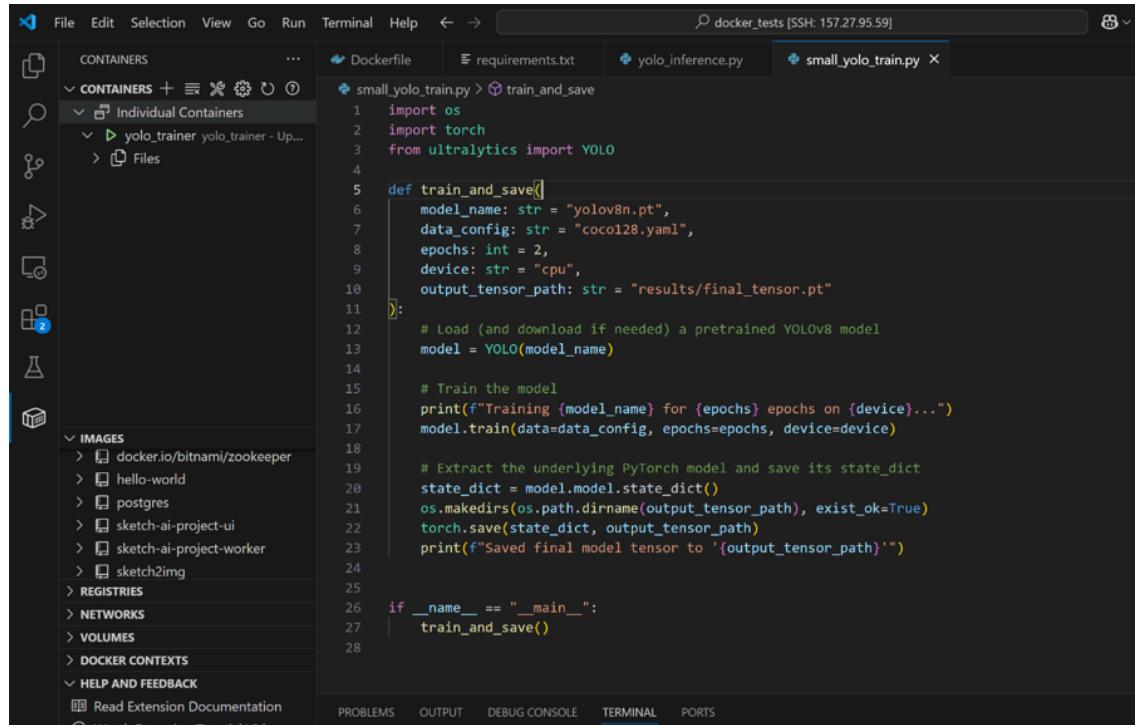
Qui vediamo che l'unica modifica è il command in fondo. E andiamo a fare:

`docker run -v 'pwd':/app test` (questo è un comando per linux, in windows dobbiamo specificare il percorso assoluto). Con questo comando mappiamo la cartella docker tests con la cartella app, facciamo mapping dell'intera cartella quindi l'eliminazione della cartella equivale alla cancellazione completa dal disco. A questo punto avvio il `docker run -v 'pwd' :/app test` ed il file verrà fisicamente buttato sul sistema op-

erativo (non dobbiamo fare mai l'intero mapping dell'intera cartella, generalmente si mappano cartelle vuote o specifiche).

## 9.5 Integrazione con VSCode del docker

Una volta sistemato tutto per farci sviluppo dobbiamo installare un'estensione per docker chiamata per l'appunto Docker. Fatto ciò ci apparirà un'icona nella barra laterale, facciamo ora un piccolo training:



```

File Edit Selection View Go Run Terminal Help < > docker_tests [SSH: 157.27.95.59]
CONTAINERS ... Dockerfile requirements.txt yolo_inference.py small_yolo_train.py
CONTAINERS + 📁 ⚙️ ⚡ ⚡️ ⓘ
Individual Containers > yolo_trainer yolo_trainer - Up...
Files

IMAGES > docker.io/bitnami/zookeeper
        > hello-world
        > postgres
        > sketch-ai-project-ui
        > sketch-ai-project-worker
        > sketch2img
REGISTRIES
NETWORKS
VOLUMES
DOCKER CONTEXTS
HELP AND FEEDBACK Read Extension Documentation
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```

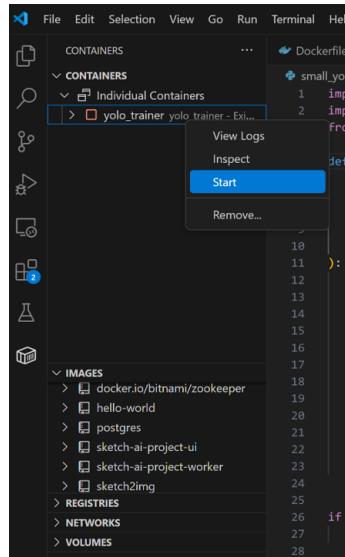
```

1 import os
2 import torch
3 from ultralytics import YOLO
4
5 def train_and_save():
6     model_name: str = "yolov8n.pt",
7     data_config: str = "coco128.yaml",
8     epochs: int = 2,
9     device: str = "cpu",
10    output_tensor_path: str = "results/final_tensor.pt"
11
12    # Load (and download if needed) a pretrained YOLOv8 model
13    model = YOLO(model_name)
14
15    # Train the model
16    print(f"Training {model_name} for {epochs} epochs on {device}...")
17    model.train(data=data_config, epochs=epochs, device=device)
18
19    # Extract the underlying PyTorch model and save its state_dict
20    state_dict = model.model.state_dict()
21    os.makedirs(os.path.dirname(output_tensor_path), exist_ok=True)
22    torch.save(state_dict, output_tensor_path)
23    print(f"Saved final model tensor to '{output_tensor_path}'")
24
25
26 if __name__ == "__main__":
27     train_and_save()
28

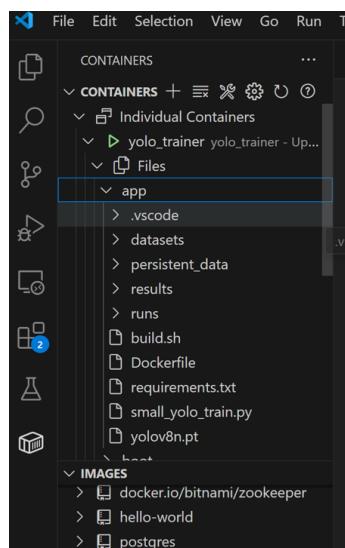
```

Questo codice è un semplice training fatto da ultralytics (package preso da Hugging Face), è un esempio diverso di training, questa volta abbiamo un pacchetto che ci permette di astrarre un pochino la parte di training di questa architettura chiamata YOLO. Detto questo parliamo di come funziona il tutto:

- Creo il codice.
- Creo il mio container.
- Quando lo voglio far partire schiaccio start:



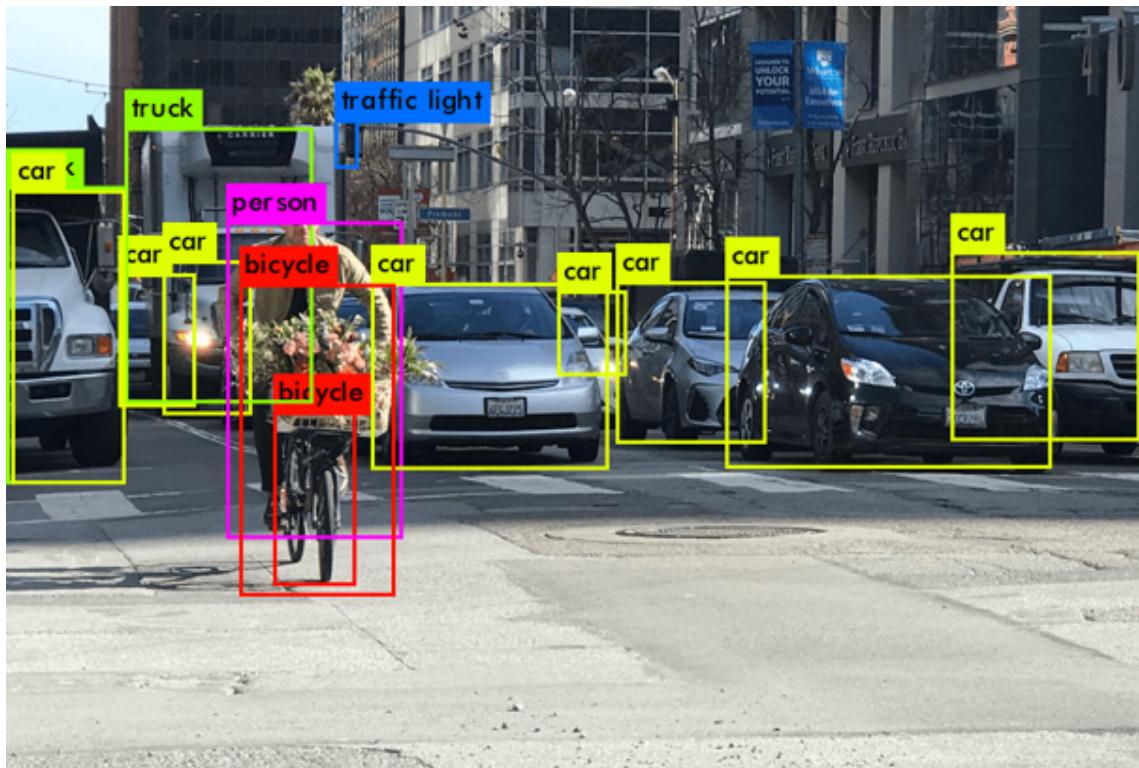
Da qui vedo in tempo reale tutto il contenuto del container.



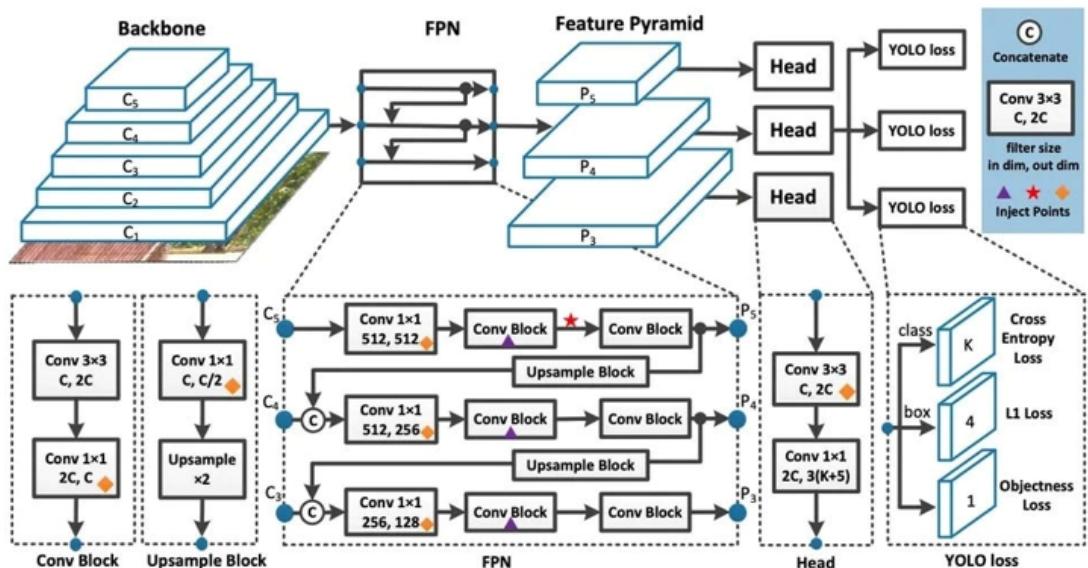
Abbiamo quindi risolto il problema ? **NO**, ci sono ancora differenze tra le versioni e le piattaforme, ci stiamo però avvicinando all'automazione delle cose. Il problema si verifica quando dobbiamo lavorare con la gpu o le reti.

## 9.6 Allenare un modello in Docker

In precedenza abbiamo visto il training tradizionale, andremo ora a trattare quello più ad alto livello, generalmente più semplice da capire per i novizi (sotto però fa le stesse identiche cose del training tradizionale). Facciamo un esempio con l'object detection, ovvero modello che ci dice quanti oggetti, dove sono e quanto spazio occupano nell'immagine. Ecco una foto di esempio:



Come esempio di dataset utilizzeremo COCO, dataset con diverse foto di migliaia di cose diverse. Un altro esempio come detto prima è YOLO, modello popolare di identificazione di oggetti. A noi interessa com'è strutturata la rete neurale, e questo è un piccolo schema:



Aggiorniamo ora i requirementes, installando ultralytics, e questo è il codice per il training (riga 13-22) ↓:

```

⌚ small_yolo_train.py > ⚗ train_and_save
1   import os
2   import torch
3   from ultralytics import YOLO
4
5   def train_and_save(
6       model_name: str = "yolov8n.pt",
7       data_config: str = "coco128.yaml",
8       epochs: int = 2,
9       device: str = "cpu",
10      output_tensor_path: str = "results/final_tensor.pt"
11  ):
12      # Load (and download if needed) a pretrained YOLOv8 model
13      model = YOLO(model_name)
14
15      # Train the model
16      print(f"Training {model_name} for {epochs} epochs on {device}...")
17      model.train(data=data_config, epochs=epochs, device=device)
18
19      # Extract the underlying PyTorch model and save its state_dict
20      state_dict = model.model.state_dict()
21      os.makedirs(os.path.dirname(output_tensor_path), exist_ok=True)
22      torch.save(state_dict, output_tensor_path)
23      print(f"Saved final model tensor to '{output_tensor_path}'")
24
25
26  if __name__ == "__main__":
27      train_and_save()
28

```

Come funziona questo codice?

- Gli diamo la versione di YOLO che vogliamo utilizzare.
- Gli diciamo che vogliamo addestrare su coco128.
- Addestro per un certo numero di epoche.
- Come device usiamo la cpu, (anche per il progetto useremo la cpu per una questione di macchina, il nostro compito per il progetto sarà capire come si fa il processo e la pipeline).
- Infine L'output tensor\_path è il modello addestrato, salvato in una cartella chiamata results.
- Carico il modello.
- Salvo con torch.save

Andiamo poi a modificare anche il dockerfile:

```
Dockerfile > ...
FROM python:3.11-slim

RUN apt-get update && apt-get install ffmpeg libsm6 libxext6 -y

# Set the working directory
WORKDIR /app
# Copy the requirements file into the container
COPY requirements.txt .
# Install the dependencies
RUN pip install --no-cache-dir -r requirements.txt
# Copy the rest of the application code into the container
COPY . .

# Run python script at container startup
# -u flag is used to force the stdout and stderr streams to be unbuffered, which can be useful for real-time logging
CMD ["python", "-u", "small_yolo_train.py"]
```

Abbiamo aggiunto una riga (riga 3), metto in coda per avere un solo layer finale.  
Poi buildo dandogli un nome ed infine facciamo il training.

Ora dobbiamo fare:

```
(base) funico@frigobar:~/docker_tests$ docker run -d -v `pwd`/persistent_data:/app/results --name yolo_trainer yolo_trainer
3b9dc98be26226f1d22a6a107a70055c8a2cf2b8d7a2ae6dae231c43ec08cc44
```

Questo perchè vogliamo che il results che salvo dentro al container finisce fuori.  
Infine lanciamo:

```
(base) fúnico@frigobar:~/docker_tests$ docker logs --follow yolo_trainer
WARNING ▲ user config directory '/root/.config/Ultralytics' is not writeable, defaulting to '/tmp' or CWD. Alternatively, Creating new Ultralytics Settings v0.0.6 file ✓
View Ultralytics Settings with 'yolo settings' or at '/tmp/Ultralytics/settings.json'
Update Settings with 'yolo settings key=value', i.e. 'yolo settings runs_dir=path/to/dir'. For help see https://docs.ultralytics.com/yolov8/
Downloading https://github.com/ultralytics/assets/releases/download/v8.3.0/yolov8n.pt to 'yolov8n.pt'...
100%|██████████| 6.25M/6.25M [00:00<00:00, 101MB/s]
Training yolov8n.pt for 2 epochs on cpu...
Ultralytics 8.3.141 ↗ Python-3.11.12 torch-2.7.0+cu126 CPU (Intel Core(TM) i7-9800X 3.80GHz)
engine/trainer: agnostic_nms=False, amp=True, augment=False, auto_augment=randaugment, batch=16, bgr=0.0, box=7.5, cache=False, copy_paste_mode=flip, cos_lr=False, cutmix=0.0, data=coco128.yaml, degrees=0.0, deterministic=True, device=cpu, dfl=False, fliplr=0.5, flipud=0.0, format=torchscript, fraction=1.0, freeze=None, half=False, hsv_h=0.015, hsv_s=0.7, hsv_v=0.01, lrf=0.01, mask_ratio=4, max_det=300, mixup=0.0, mode=train, model=yolov8n.pt, momentum=0.937, mosaic=1.0, multi_label=True, overlap_mask=True, patience=100, perspective=0.0, plots=True, pose=12.0, pretrained=True, profile=False, project=None,rop=False, save_dir=runs/detect/train, save_frames=False, save_json=False, save_period=1, save_txt=False, scale=0.5, splitify=True, single_cls=False, source=None, split=val, stream_buffer=False, task=detect, time=None, tracker=botsort.yaml, lr=0.1, warmup_epochs=3.0, warmup_momentum=0.8, weight_decay=0.0005, workers=8, workspace=None

WARNING ▲ Dataset 'coco128.yaml' images not found, missing path '/app/datasets/coco128/images/train2017'
Downloading https://ultralytics.com/assets/coco128.zip to '/app/datasets/coco128.zip'...
100%|██████████| 6.66M/6.66M [00:00<00:00, 98.8MB/s]
Unzipping /app/datasets/coco128.zip to /app/datasets/coco128...: 100%|██████████| 263/263 [00:00<00:00, 5470.51file/s]
Dataset download success ✓ (0.8s), saved to /app/datasets

Downloading https://ultralytics.com/assets/Arial.ttf to '/tmp/Ultralytics/Arial.ttf'...
100%|██████████| 755k/755k [00:00<00:00, 46.0MB/s]

      from    n   params  module                                     arguments
0           -1    1       464  ultralytics.nn.modules.conv.Conv          [3, 16, 3, 2]
1           -1    1      4672  ultralytics.nn.modules.conv.Conv          [16, 32, 3, 2]
2           -1    1      7360  ultralytics.nn.modules.block.C2f          [32, 32, 1, True]
```

Quando finisce vedremo le performance:

```

train: Fast image access ✓ (ping: 0.0±0.0 ms, read: 1621.3±565.2 MB/s, size: 50.9 KB)
train: Scanning /app/datasets/coco128/labels/train2017... 126 images, 2 backgrounds, 0 corrupt: 100%|██████████| 128/128 [00:00<0:00, 2091.12it/s]
train: New cache created: /app/datasets/coco128/labels/train2017.cache
/usr/local/lib/python3.11/site-packages/torch/utils/data/dataloader.py:665: UserWarning: 'pin_memory' argument is set as true but no accelerator is
    warnings.warn(warn_msg)
val: Fast image access ✓ (ping: 0.0±0.0 ms, read: 1860.4±707.2 MB/s, size: 52.5 KB)
val: Scanning /app/datasets/coco128/labels/train2017.cache... 126 images, 2 backgrounds, 0 corrupt: 100%|██████████| 128/128 [00:00<?, ?it/s]
/usr/local/lib/python3.11/site-packages/torch/utils/data/dataloader.py:665: UserWarning: 'pin_memory' argument is set as true but no accelerator is
    warnings.warn(warn_msg)
Plotting labels to runs/detect/train/labels.jpg...
optimizer: 'optimizer-auto' found, ignoring 'lr=0.01' and 'momentum=0.937' and determining best 'optimizer', 'lr0' and 'momentum' automatically...
optimizer: AdamW(lr=0.000119, momentum=0.9) with parameter groups 57 weight(decay=0.0), 64 weight(decay=0.0005), 63 bias(decay=0.0)
Image sizes 640 train, 640 val
Using 0 dataloader workers
Logging results to runs/detect/train
Starting training for 2 epochs...

```

Epoch	GPU_mem	box_loss	cls_loss	dfl_loss	Instances	Size
1/2	0G	1.178	1.565	1.237	285	640: 100% ██████████  8/8 [00:19<00:00, 2.43s/it]
	Class	Images	Instances	Box(P R		mAP50 mAP50-95: 100% ██████████  4/4 [00:06<00:00, 1.75s/it]
	all	128	929	0.661	0.52	0.612 0.454

Epoch	GPU_mem	box_loss	cls_loss	dfl_loss	Instances	Size
2/2	0G	1.19	1.387	1.261	256	640: 100% ██████████  8/8 [00:17<00:00, 2.19s/it]
	Class	Images	Instances	Box(P R		mAP50 mAP50-95: 100% ██████████  4/4 [00:06<00:00, 1.64s/it]
	all	128	929	0.668	0.532	0.622 0.462

2 epochs completed in 0.014 hours.  
Optimizer stripped from runs/detect/train/weights/last.pt, 6.5MB  
Optimizer stripped from runs/detect/train/weights/best.pt, 6.5MB

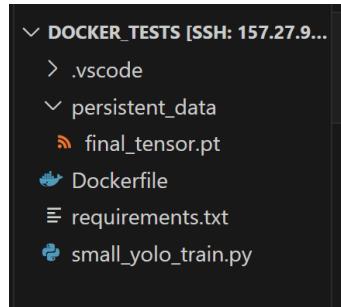
Validating runs/detect/train/weights/best.pt...

Ultralytics 8.3.141 🚀 Python-3.11.12 torch-2.7.0+cu126 CPU (Intel Core(TM) i7-9800X 3.80GHz)

Model summary (fused): 72 layers, 3,151,904 parameters, 0 gradients, 8.7 GFLOPs

Class	Images	Instances	Box(P R	mAP50 mAP50-95:	100% ██████████  4/4 [00:05<00:00, 1.35s/it]
all	128	929	0.666 0.532	0.622 0.463	
person	61	254	0.813 0.667	0.772 0.542	
bicycle	3	6	0.55 0.333	0.325 0.282	
car	12	46	0.763 0.217	0.282 0.178	
motorcycle	4	5	0.681 0.86	0.898 0.72	
airplane	5	6	0.84 0.88	0.948 0.695	
bus	5	7	0.657 0.714	0.73 0.676	
train	3	3	0.544 0.667	0.741 0.61	
truck	5	12	1 0.33	0.521 0.304	
boat	2	6	0.432 0.265	0.399 0.279	

Quando finisce tutto vado a cercare il mio dato con il comando `docker ps -a`



## 9.7 Inferenza

L'inferenza è il processo mediante il quale un modello di machine learning già addestrato viene utilizzato per fare previsioni o classificazioni su nuovi dati mai visti prima. In pratica, dopo aver completato la fase di training e aver salvato i pesi del modello, si carica il modello e si forniscono in input nuovi dati per ottenere un output predittivo.

Nel contesto di un modello di deep learning (ad esempio una rete neurale PyTorch), il flusso tipico di inferenza è:

- Caricamento del modello addestrato:** si caricano i pesi salvati tramite `model.load_state_dict()`.

2. **Preparazione dei dati di input:** i dati devono essere preprocessati nello stesso modo usato durante il training (es. normalizzazione, ridimensionamento, conversione in tensori).
3. **Impostazione del modello in modalità valutazione:** si usa `model.eval()` per disabilitare comportamenti specifici del training come dropout e batch normalization adattativa.
4. **Esecuzione della predizione:** si passa il dato al modello e si ottiene l'output, tipicamente all'interno di un blocco `with torch.no_grad()`: per evitare il calcolo dei gradienti.

**Esempio pratico in PyTorch:**

```
import torch
from src.train_mnist import SimpleCNN

# Carica il modello addestrato
model = SimpleCNN()
model.load_state_dict(torch.load("mnist_model.pth"))
model.eval()

# Prepara un'immagine di input (ad esempio una singola immagine MNIST)
# Supponiamo che 'image' sia un tensore di forma [1, 28, 28]
image = ... # Carica e preprocessa l'immagine

# Aggiungi la dimensione batch e passa in float32
input_tensor = image.unsqueeze(0).float()

# Esegui l'inferenza
with torch.no_grad():
    output = model(input_tensor)
    predicted_class = output.argmax(dim=1).item()

print(f"Classe predetta: {predicted_class}")
```

**Nota:** In produzione, l'inferenza viene spesso eseguita in ambienti containerizzati (ad esempio tramite Docker) per garantire portabilità e riproducibilità del risultato.

**In sintesi:** l'inferenza è la fase in cui il modello, già addestrato, viene utilizzato per generare previsioni su nuovi dati, rappresentando il vero valore applicativo del machine learning.

## 9.8 CI/CD con Docker

Ovviamente si può integrare Docker nel nostro CD/CI, tendenzialmente sono molto pesanti quindi come detto prima si consiglia di partire da cose piccole. Questo è un esempio di CI/CD con Docker:

```
- name: Build Docker image
  run: |
    docker build -t your-dockerhub-username/your-app:latest .

# Login to Docker Hub
- name: Login to Docker Hub
  uses: docker/login-action@v3
  with:
    username: ${{ secrets.DOCKER_HUB_USERNAME }}
    password: ${{ secrets.DOCKER_HUB_PASSWORD }}

# Push Docker image to Docker Hub
- name: Push Docker image
  run: |
    docker push your-dockerhub-username/your-app:latest

# Save the Docker image as an artifact (tar file)
- name: Save Docker image as artifact
  run: |
    docker save your-dockerhub-username/your-app:latest -o your-app-latest.tar
- name: Upload Docker image artifact
  uses: actions/upload-artifact@v4
  with:
    name: docker-image
    path: your-app-latest.tar
```

## 10 Logging e monitoring

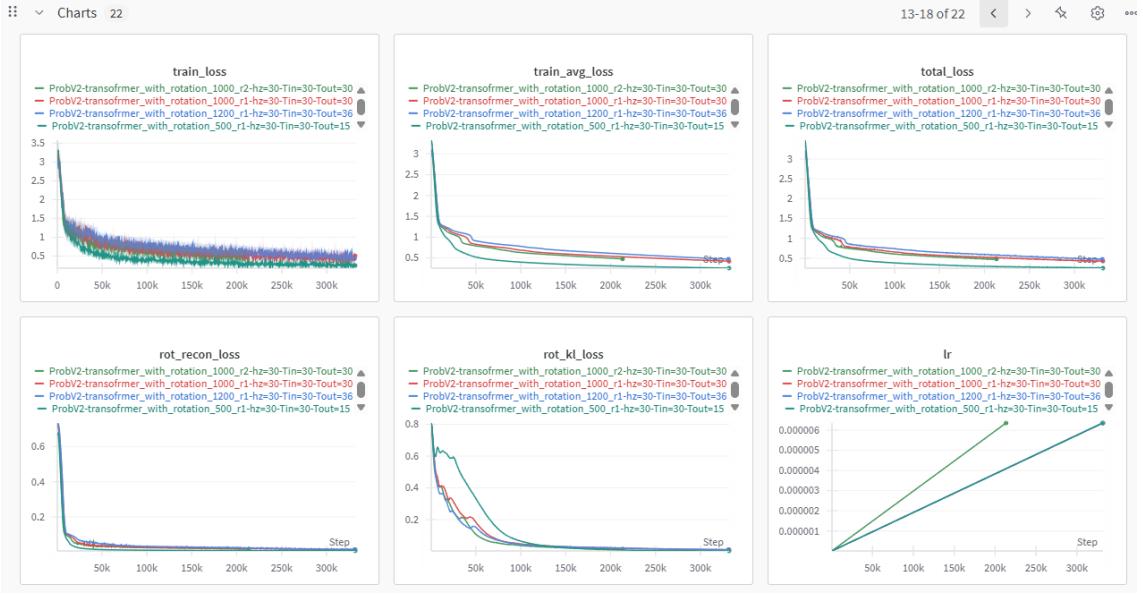
Nel logging ci sono 3 livelli:

1. Livello sperimentale.
2. Livello strutturale.
3. Livello log aggregation.

A livello di training ci sono diversi modi di loggare le informazioni, Wights & Biases è un ottimo strumento per questo.

```
import wandb
# 1. Start a new run
run = wandb.init(project="gpt5")
# 2. Save model inputs and hyperparameters
config = run.config
config.dropout = 0.01
# 3. Log gradients and model parameters
run.watch(model)
for batch_idx, (data, target) in enumerate(train_loader):
    ...
    if batch_idx % args.log_interval == 0:
        # 4. Log metrics to visualize performance
        run.log({"loss": loss})
```

Questa è la parte di training, come vediamo ogni tanto viene fatto un log, con wandb, una volta inizializzato un progetto e fatto tutti i cicli ci stampa, invece che sulla riga di comando diversi grafici tramite interfaccia grafica.



Naturalmente possiamo decidere noi cosa loggare. Possiamo farlo anche con le matrici di confusione.

## 10.1 Log aggregation and training

Ci permette di centralizzare gli standard output ma anche gli errori, permettendo di avere un log anche quando sono distribuite le macchine. Con questa cosa qui, se ci sono 100 macchine ed una va in errore, c'è un punto centralizzato dove ti arrivano tutte le macchine e tutti gli errori. Tutto basato su text. Le sfide di questo log aggregation sono che dobbiamo gestire diverse fonti di dati differenti, l'altra cosa è che abbiamo una grossa quantità di dati e se viene fatto su cloud costa. Un esempio è **sentry**:

The screenshot shows the Sentry Python monitoring interface. It features a Python logo icon and the title "Python Error and Performance Monitoring". Below the title, there's a brief description: "Actionable insights to resolve Python performance bottlenecks and errors. See the full picture of any Python exception so you can diagnose, fix, and optimize performance in the Python debugging process." At the bottom, there are two buttons: "TRY SENTRY FOR FREE" and "REQUEST A DEMO". On the right side, there's a panel titled "Grab the Sentry Python SDK:" with a code snippet: 

```
pip install --upgrade sentry-sdk
```

. Below it, another panel titled "Configure your DSN:" contains a code snippet for initializing the SDK: 

```
import sentry_sdk
sentry_sdk.init(
    "https://<key>@sentry.io/<project>",

    # Set traces_sample_rate to 1.0 to capture 100%
    # of transactions for Tracing.
    # We recommend adjusting this value in production.
    enable_tracing=True,
    traces_sample_rate=1.0,
)
```

Una volta inizializzata il logging avviene automaticamente. Avremo quindi un debugging di qualità ed intuitivo. Ecco un esempio:

```
(.venv) => ~/dev/devtech-solutions-python => python main.py
=====
Running DevTech Solutions super mega algorithm scripty thing =====
Traceback (most recent call last):
  File "/Users/salma/dev/devtech-solutions-python/main.py", line 16, in <module>
    division_by_zero = 1 / 0
                    ~~^~~
ZeroDivisionError: division by zero
(.venv) => ~/dev/devtech-solutions-python =>
```

Unendo il tutto a sentry avremo una pagina web che ci dirà tutti i problemi. ↓

The screenshot shows a Sentry interface for a ZeroDivisionError event. The event details are as follows:

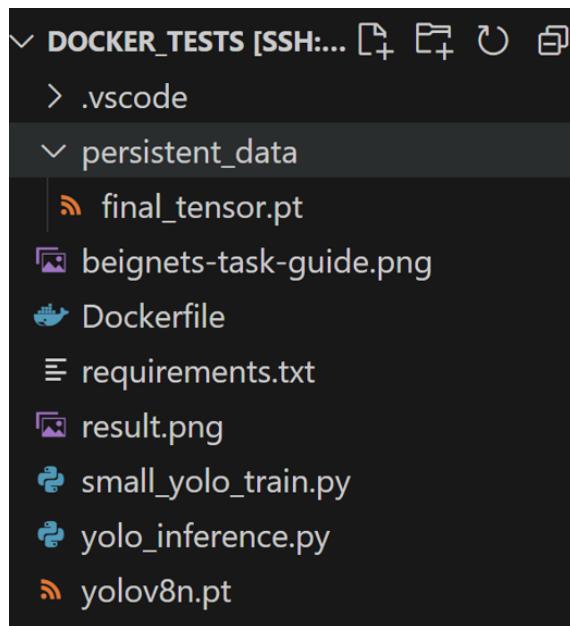
- Event ID:** 1ba724d4 Dec 19, 9:45 AM
- Root Cause:** Unhandled division by zero
- Tags:** environment: production, handled: no, level: error, mechanism: excepthook, runtime: CPython 3.11.6, runtime.name: CPython
- Stack Trace:**

```
main.py in <module> at line 16
11     profiles_sample_rate=1.0,
12 )
13
14     print("== Running DevTech Solutions super mega algorithm scripty thing ==")
15
16     division_by_zero = 1 / 0
17
...Annotations... {}
```
- Assigned To:** No one
- Last 24 Hours:** 1
- Last 30 Days:** 1
- Last Seen:** 2 minutes ago
- First Seen:** 2 minutes ago
- Releases:** See which release caused this issue
- Issue Tracking:** Track this issue in Jira, GitHub, etc.

## 11 Python Packaging

Essendo python un interprete ci sono diversi problemi nell'import ricorrenti o ricorsivi, il discorso delle doppie dipendenze può portare ad errori. Stesso discorso con gli import relativi, per ovviare a questi problemi si indica, all'avvio del programma, la cartella di partenza.

L'ultima volta eravamo rimasti qua:



Per strutturare tutto questo in un pacchetto il team di python ha deciso di mettere tutti i sorgenti dentro la cartella **src**, dobbiamo creare un pacchetto con un nome, in questo caso si chiama `docker_tests`, e dentro una cartella `src`.

```
▽ DOCKER_TESTS [SSH:...]
  ▽ .github/workflows
    ! deploy.yml
  > .pytest_cache
  ▽ .vscode
    { launch.json
  > data
  ▽ persistent_data
    final_tensor.pt
  ▽ src
    > __pycache__
    eda.py
    small_yolo_train.py
    train_mnist.py
    yolo_inference.py
```



```
▽ tests
  > __pycache__
  test_eda.py
  test_train.py
  test_yolo.py
  > venv
  .coverage
  .gitignore
  beignets-task-guide.png
  bus.jpg
  Dockerfile
  Makefile
  requirements.txt
  result.png
  yolov8n.pt
```

Fatto questo dobbiamo scrivere i test ed aggiornare il .gitignore.

```

.gitignore
1   .pytest_cache/
2   venv/
3   .coverage
4
5   *.pyc
6   __pycache__/
7   *.pyo
8   *.pyd
9
10  data/
11  *.pth
12  *.pt
13
14  persistent_data/
15
16  *.png
17  *.jpg
18  *.jpeg

```

Aggiorniamo un minimo il dockerfile:

```

Dockerfile > ...
1  FROM python:3.11-slim
2
3  RUN apt-get update && apt-get install ffmpeg libsm6 libxext6 -y
4
5  # Set the working directory
6  WORKDIR /app
7  # Copy the requirements file into the container
8  COPY requirements.txt .
9  # Install the dependencies
10 RUN pip install --no-cache-dir -r requirements.txt
11 ## To answer the question of the last time about the requirements.txt file,
12 # we are copying it before so that we can cache the layer with the dependencies.
13
14 # Copy the rest of the application code into the container
15 COPY .
16
17 # Run python script at container startup
18 # -u flag is used to force the stdout and stderr streams to be unbuffered, which can be useful for real-time logging
19 CMD ["python", "-u", "src/small_yolo_train.py"]
20 |

```

Creiamo ora il pacchetto:

- è un modo per organizzare il nostro progetto in modo che le persone possano andare a prenderlo e scaricarlo senza vedere com'è strutturato tutto il progetto.
- è utile creare un pacchetto finale del nostro progetto.

Ma come si fa a creare un pacchetto python?

- Il primo modo è quello di creare un file chiamato setup.py e qui definiamo lato codice "creami il pacchetto"
- Un altro modo più veloce e moderno è tramite software esterni come toml:

```

❶ pyproject.toml
1   [build-system]
2     requires = ["setuptools>=61.0", "wheel"]
3     build-backend = "setuptools.build_meta"
4
5   [project]
6     name = "mlops-test"
7     version = "0.1.0"
8     description = "A short description of your mlops project"
9     authors = [
10       { name = "Your Name", email = "you@example.com" }
11     ]
12     readme = "README.md"
13     requires-python = ">=3.10"
14     license = { text = "MIT" }
15
16   dependencies = [
17   ]
18
19   [project.urls]
20     Homepage = "https://github.com/yourusername/your-project"
21     Documentation = "https://github.com/yourusername/your-project/wiki"
22

```

**Poetry** è uno strumento moderno per la gestione delle dipendenze e la creazione di pacchetti Python. Rispetto ai metodi tradizionali (`setup.py`, `requirements.txt`), Poetry offre:

- Gestione automatica delle dipendenze (inclusi sub-dependencies)
- Ambiente virtuale isolato per ogni progetto
- Comandi semplici per build, pubblicazione e gestione versioni
- Un unico file di configurazione: `pyproject.toml`

#### Installazione:

```
pip install poetry
```

#### Creazione di un nuovo progetto:

```
poetry new nome_progetto
```

#### Aggiunta di una dipendenza:

```
poetry add numpy
```

#### Attivazione dell'ambiente virtuale:

```
poetry shell
```

## Build e pubblicazione:

```
poetry build  
poetry publish
```

Poetry semplifica la gestione dei progetti Python, rendendo più facile mantenere un ambiente riproducibile e pronto per la distribuzione. Fatta la gestione delle dipendenze risulterà una cosa simile:

```
❶ pyproject.toml  
1   [build-system]  
2     requires = ["setuptools>=61.0", "wheel"]  
3     build-backend = "setuptools.build_meta"  
4  
5   [project]  
6     name = "mlops"  
7     version = "0.1.0"  
8     description = "A short description of your mlops project"  
9     authors = [  
10       { name = "Your Name", email = "you@example.com" }  
11     ]  
12     readme = "README.md"  
13     requires-python = ">=3.10"  
14     license = { text = "MIT" }  
15  
16   dependencies = [  
17     "pylint (>=3.3.7,<4.0.0)",  
18     "pytest (>=8.3.5,<9.0.0)",  
19     "pytest-cov (>=6.1.1,<7.0.0)",  
20     "ultralytics (>=8.3.146,<9.0.0)",  
21     "torch (>=2.7.0,<3.0.0)",  
22     "torchvision (>=0.22.0,<0.23.0)",  
23     "scikit-learn (>=1.6.1,<2.0.0)",  
24     "matplotlib (>=3.10.3,<4.0.0)",  
25     "seaborn (>=0.13.2,<0.14.0)",  
26     "tqdm (>=4.67.1,<5.0.0)"  
27   ]  
28  
29   [project.urls]  
30     Homepage = "https://github.com/yourusername/your-project"  
31     Documentation = "https://github.com/yourusername/your-project/wiki"  
32   |
```

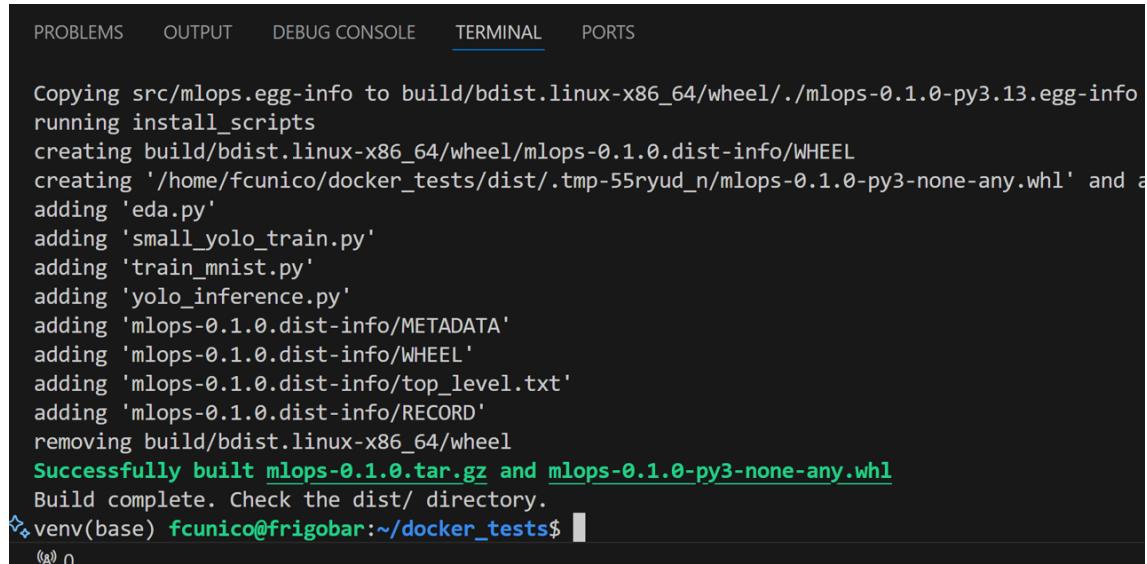
Il makefile aggiornato sarà così:

```

M Makefile
1  install:
2      python -m pip install --upgrade pip
3      python -m pip install -r requirements.txt
4      @echo "Installation complete. You can now run the project."
5
6  lint:
7      pylint --disable=R,C src/*.py tests/*.py
8      @echo "Linting complete."
9
10 test:
11     python -m pytest -vv --cov=src tests/
12     @echo "Testing complete."
13
14 build:
15     python -m build
16     @echo "Build complete. Check the dist/ directory."
17
18 clean:
19     rm -rf dist/ build/ *.egg-info .pytest_cache __pycache__
20     @echo "Clean complete."
21

```

A questo punto facciamo build:



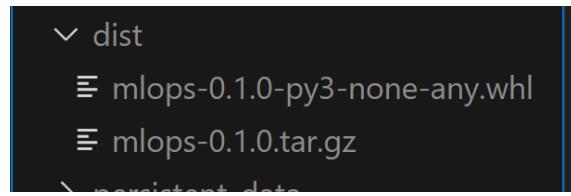
```

PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL    PORTS

Copying src/mlops.egg-info to build/bdist.linux-x86_64/wheel./mlops-0.1.0-py3.13.egg-info
running install_scripts
creating build/bdist.linux-x86_64/wheel/mlops-0.1.0.dist-info/WHEEL
creating '/home/funico/docker_tests/dist/.tmp-55ryud_n/mlops-0.1.0-py3-none-any.whl' and a
adding 'eda.py'
adding 'small_yolo_train.py'
adding 'train_mnist.py'
adding 'yolo_inference.py'
adding 'mlops-0.1.0.dist-info/METADATA'
adding 'mlops-0.1.0.dist-info/WHEEL'
adding 'mlops-0.1.0.dist-info/top_level.txt'
adding 'mlops-0.1.0.dist-info/RECORD'
removing build/bdist.linux-x86_64/wheel
Successfully built mlops-0.1.0.tar.gz and mlops-0.1.0-py3-none-any.whl
Build complete. Check the dist/ directory.
$ venv(base) funico@frigobar:~/docker_tests$ 

```

Ora si può installare con `pip install mlops-0.1.0-py3-none-any-whl`:



Una volta creato il tomli facendo `pip install`. lui farà tutto. Questi sono degli artifacti, oggetti che vengono prodotti alla fine dell'operazione di build e che possiamo

rendere disponibili agli altri utenti.  
Ecco un esempio di artifact:

```
1  name: Build and Deploy
2  on:
3    push:
4      branches:
5        - main
6  jobs:
7    build-and-deploy:
8      runs-on: ubuntu-latest
9
10   steps:
11     # Checkout repo
12     - name: Checkout code
13       uses: actions/checkout@v3
14
15     # Setup Python
16     - name: Setup Python
17       uses: actions/setup-python@v4
18       with:
19         python-version: "3.10"
20
21     # Install build dependencies
22     - name: Install build dependencies
23       run: |
24         make install
25         pip install build
26
27     # Build the wheel and tar.gz using Makefile
28     - name: Build Python package
29       run: make build
30
31     # Upload built artifacts (.whl, .tar.gz)
32     - name: Upload Python package artifacts
33       uses: actions/upload-artifact@v4
34       with:
35         name: python-package
36         path: dist/*
```

- checkout
- scarica la repository
- setup dei file
- build delle dipendenze (make install)
- operazioni di build
- carichiamo il python package con tutto quello che c'è nella cartella dist

Ecco un esempio di deploy di modello addestrato:

```

1  name: YOLO Trainer Docker Build & Artifact Upload
2
3  on:
4    push:
5      branches:
6        - main
7
8  jobs:
9    train-and-upload:
10      runs-on: ubuntu-latest
11
12    steps:
13      # Checkout repo
14      - name: Checkout code
15      | uses: actions/checkout@v3
16
17      # Build Docker image
18      - name: Build YOLO trainer Docker image
19      | run: |
20      |   docker build -t yolo_trainer .
21
22      # Run the Docker container to train
23      - name: Run training container
24      | run: |
25      |   docker run --name yolo_trainer_container yolo_trainer
26
27      # Extract the final_tensor.pt file from the container
28      - name: Copy final_tensor.pt from container
29      | run: |
30      |   docker cp yolo_trainer_container:/app/results/final_tensor.pt final_tensor.pt
31
32      # Upload final_tensor.pt as artifact
33      - name: Upload trained model artifact
34      | uses: actions/upload-artifact@v4
35      | with:
36      |   name: trained-model
37      |   path: final_tensor.pt
38
39      # Clean up container
40      - name: Remove container
41      | run: |
42      |   docker rm yolo_trainer_container
43

```

Attenzione allo spazio degli artifatti, le repository pubbliche non hanno un limite, quelle private hanno 500MB.

## 12 AutoML

L'AutoML (Automated Machine Learning) è un insieme di tecniche e strumenti che automatizzano le fasi principali del ciclo di vita di un progetto di machine learning, come la selezione del modello, la scelta e l'ottimizzazione degli iperparametri, la preparazione dei dati e la valutazione delle performance. L'obiettivo è rendere accessibile il machine learning anche a chi non è esperto, riducendo il tempo e la complessità necessari per sviluppare modelli efficaci.

### 12.1 Caratteristiche principali di AutoML

- **Preprocessing automatico dei dati:** gestione di dati mancanti, normalizzazione, encoding delle variabili categoriche.
- **Selezione automatica dei modelli:** esplorazione di diversi algoritmi (es. alberi decisionali, reti neurali, SVM) per trovare quello più adatto al problema.
- **Ottimizzazione degli iperparametri:** ricerca automatica dei parametri migliori tramite tecniche come grid search, random search o ottimizzazione bayesiana.
- **Valutazione e selezione:** confronto tra modelli tramite metriche di performance e selezione automatica del migliore.
- **Deployment facilitato:** alcuni strumenti AutoML permettono di esportare direttamente il modello pronto per la produzione.

### 12.2 Vantaggi e limiti

**Vantaggi:**

- Riduce la necessità di competenze avanzate di ML.
- Accelera il processo di sviluppo.
- Permette di testare rapidamente molte soluzioni.

**Limiti:**

- Meno controllo sui dettagli del modello.
- Può essere computazionalmente costoso.
- Non sostituisce l'esperienza umana nell'interpretazione dei risultati e nella gestione dei dati.

## 12.3 Esempi di strumenti AutoML

- **Auto-sklearn:** estensione di scikit-learn per la selezione automatica di modelli e iperparametri.
- **TPOT:** utilizza algoritmi genetici per ottimizzare pipeline di machine learning.
- **H2O AutoML:** piattaforma open source per la creazione automatica di modelli.
- **Google Cloud AutoML, Azure AutoML, Amazon SageMaker Autopilot:** soluzioni cloud che offrono pipeline AutoML complete.

## 12.4 Esempio pratico con Auto-sklearn

```
import autosklearn.classification
from sklearn.datasets import load_digits
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

# Carica dati di esempio
X, y = load_digits(return_X_y=True)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)

# Crea e addestra AutoML classifier
automl = autosklearn.classification.AutoSklearnClassifier(
    time_left_for_this_task=60)
automl.fit(X_train, y_train)

# Valuta il modello
y_pred = automl.predict(X_test)
print("Accuracy:", accuracy_score(y_test, y_pred))
```

In questo esempio, Auto-sklearn seleziona e ottimizza automaticamente il modello migliore per il dataset delle cifre scritte a mano, mostrando la semplicità d'uso di uno strumento AutoML.

## 13 Tool utili

Partiamo da Gradio, interfaccia web che ci permette di gestire input ed output di reti neurali, ha un sacco di strumenti ed accessori per la modifica del progetto. Si importa così:

```

import gradio as gr

def greet(name):
    return "Hello " + name + "!"

demo = gr.Interface(fn=greet, inputs="text", outputs="text")
demo.launch()

```

The screenshot shows a Gradio interface for a simple 'greet' function. The interface consists of two text input fields: 'name' and 'output'. Below these fields are three buttons: 'Clear', 'Submit', and 'Share via Link'. At the bottom of the interface, there is a message indicating it was 'built with Gradio.' and 'Hosted on Spaces'.



Definiamo una funzione, in questo caso un'interfaccia, input e output testuali ed infine facciamo launch. Un'implementazione con YOLO è:

```

src > ✎ gradio_example.py > ...
  1  import gradio as gr
  2  from ultralytics import YOLO
  3
  4  # Load the YOLO model
  5  model = YOLO("yolov8n.pt")  # You can use your custom model path if needed
  6
  7  def detect_objects(image):
  8      # Inference
  9      results = model(image)
 10
 11      # The first result is usually the full prediction
 12      annotated_frame = results[0].plot()  # Draws bounding boxes
 13
 14      return annotated_frame
 15
 16  # Gradio interface
 17  iface = gr.Interface(
 18      fn=detect_objects,
 19      inputs=gr.Image(type="filepath", label="Upload Image"),
 20      outputs=gr.Image(label="Detected Objects"),
 21      title="YOLO Object Detection",
 22      description="Upload an image and let YOLO detect objects in it!"
 23  )
 24
 25  if __name__ == "__main__":
 26      iface.launch()
 27

```

## 14 Progetto d'esame

Fatto il progetto dobbiamo inviare il link github (scrivendoci un minimo di documentazione), devono esserci le github action e ci deve essere un minimo di storia (viene valutato l'utilizzo degli strumenti), bisogna usare il linter e unit test. Il progetto sarà valutato sui concetti visti a lezione di DevOps:

- Python DevOps (git, lin, unit tests, artifacts);
- Dockerizzazione;
- Training e testing di un piccolo modello;
- Github actions per CI/CD;

Come sorgente di dataset possiamo scegliere quello che vogliamo, se usiamo dataset di immagini scegliamo tra le mille e le trentamila, non di più.

Il corso è superato sulla base di un punteggio assegnato al progetto con un minimo di 6.0 punti, con possibili punti bonus extra: Il punteggio è così diviso:

- Gt repository (pubblico e package python): fino a 2 punti
- EDA su un dataset a scelta (consiglio piccolo): fino a 2 punti
- Dockerizzazione della soluzione, compreso addestramento: fino a 4 punti
- CI/CD (github actions): unit, linter, docker build, artifact release: fino a 2 punti
- Opzionale:
  - docler image push su docker hub (pubblico): fino a 1 punto
  - interfaccia interattiva con Gradio correttamente dockerizzata: fino a 1 punto

Nota importante, del progetto verrà valutata in modo preciso l'organizzazione ed il funzionamento, le performance conteranno poco viste le macchine limitate.