

Deep Learning Pipeline on Histopathology Images


Detection of Prostatic Tumor

F. VERSACI G. BUSONERA

Scalable Computing and Analytics Group
CRS4, Cagliari, Italy

– DeepHealth* Winter School 2022 –

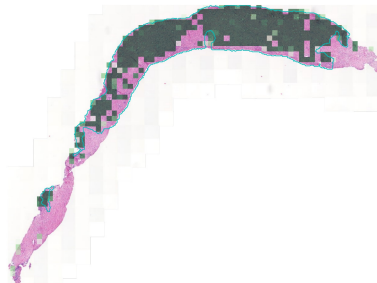


*  H2020 funded project with agreement number 825111

- Glass slides containing **tissue from biopsy** are now digitized to produce **whole slide images (WSI)**
- Automatic classification of tissue to help **tumor detection** is an important problem in **Digital Pathology**
- It can be a **powerful diagnostic decision support tool** to help and **speed up** the job of **pathologists**



Example of WSI

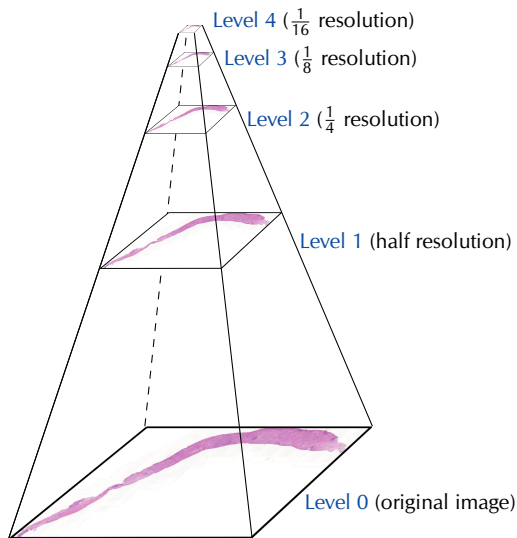


Pyramidal images

Five-level example



DEEPHEALTH



Pyramidal storage

Data are stored

- in a **tiled** fashion, to allow **quick panning**
- in **multiple resolutions**, to allow **quick zooming**

- Provided by the **Karolinska Institutet**
- The **raw input dataset** consists of hundreds of **gigapixel slides**
- **Small patches** are extracted from the slides
- Each patch has a **label** (or more) and some **metadata** (e.g., coordinates, anonymized patient ID, date, etc.)

Number of cases – slides: 200 – 384 (i.e., about 2 slides/case)

Size of slides: $112,908 \times 264,186$ (\simeq **30 gigapixels/slide**)

Tissue percentage: less than 3% per slide

Average size on disk: 660 MB/slide (MIRAX+JPEG)

Annotated tissue: about 0.35% per slide

Size of patches: 256×256 (at **zoom level 1**)

Number of extracted patches: 123,148 (training + validation)

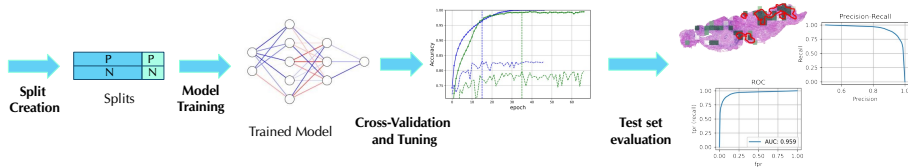
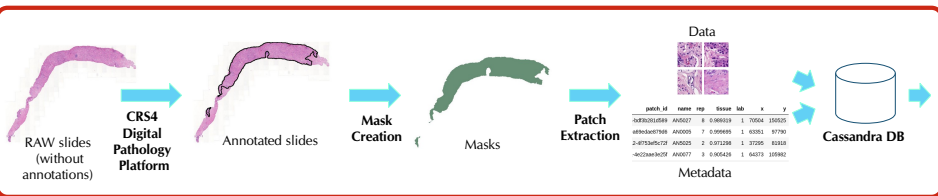
Normal vs tumoral patches: 37,695 vs 85,453

Full predictive pipeline

Bird's-eye view



DEEPHEALTH

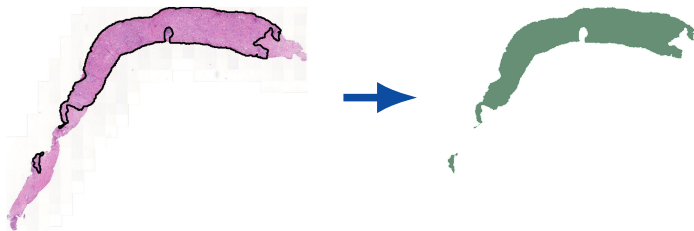


Input – from the Digital Pathology Platform

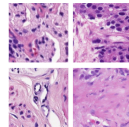
Slides: **MIRAX+JPEG** format

Annotations: paths encoded as **CSV+JSON**

- Conversion of annotations to **PNG masks at level 6**
- **1 pixel (level 6) = 64×64 pixels (level 0)**
- Output will be: **normal** mask + **tumoral** mask
- **Python** program using PIL and OpenSlides
- Parallelized via **Apache Spark**



- Rescale masks at **level 9**
- **1 pixel (level 9) = 256×256 pixels (level 1)**
- For each mask point extract the corresponding patch at level 1
- **Filter out** patches with less than **30% tissue**
- Parallelized via **Apache Spark**



patch_id	name	rep	tissue	lab	x	y
-bdf3b281d589	AN5027	8	0.989319	1	70504	150525
a69edae879d6	AN0005	7	0.999695	1	63351	97790
2-4f753ef5c72f	AN5025	2	0.971298	1	37295	81918
-4e22aae3e25f	AN0077	3	0.905426	1	64373	105982

Where do we **save patches** and **metadata**?

- Many **small files** saved in a **filesystem**
- Path/names encoding split and class (e.g., validation/tumor)
- Other **metadata** saved in external table (e.g., CSV file)

Usability issues

- The dataset is **static**
- Changes (e.g, a different train/validation ratio) may require **recreating or moving many files**
- **Data and metadata** are **decoupled**, which can more easily lead to **inconsistencies**

Performance issues

- **Accessing many small files** is a known **weak point** for filesystems
- To allow parallel access:
Network storage Not scalable, bottlenecks on networks and disks
Parallel filesystem Complex to configure

- Scalable and low-latency network access to DL datasets
- Random access to stored images
- Coupled data and metadata storage
- Dynamic and easy split creation
- Simplify data distribution in parallel DL training

Workflow

- Images are saved as BLOBs in a Cassandra DB
- Labels and metadata are saved together with the images
- Each image is identified by a UUID
- The splits (lists of UUIDs) are created automatically, based on target values and constraints involving metadata
- When needed, data are efficiently fetched by their UUID, and fed to the DL library

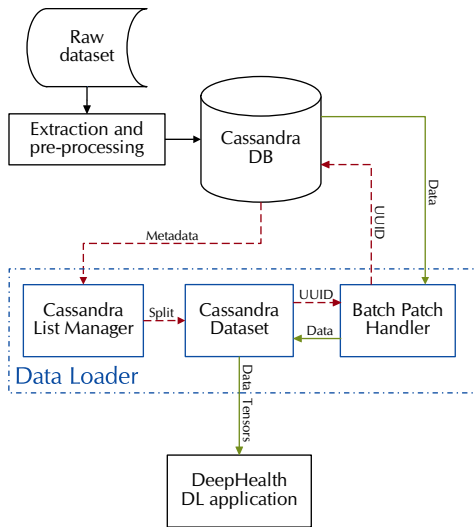
DeepHealth Toolkit

- **Open-source** DL toolkit
- Particularly focused on enabling easy DL adoption in the **medical field**
- Written in C++, exposes **C++ and Python APIs**

Apache Cassandra

- **Distributed**, decentralized and **highly scalable NoSQL DB**
- **Free** and open-source project
- Widely adopted both in **industry** and **big data analytics** contexts
- **Low latency** (typically less than a millisecond)

- A similar architecture could be implemented also adopting **different DBs**, provided they offer **horizontal scalability** and **low latency access**



- Images are extracted from the **raw dataset**
- They are **pre-processed and inserted** (together with relevant **metadata**) in the **Cassandra DB**
- The DB is queried to build the **list of splits**
- Finally, the **images and labels** are fetched **whenever needed** by the DL application

- Contains all the **metadata** and a **randomly generated UUID**
- Its **partition keys** are the “**natural**” ones of the dataset
- Used when **creating the splits**
- It allows, e.g., to fetch the UUIDs of the patches for **any given patient/slide/label combination**

```
CREATE TABLE patches.metadata_by_nat(  
  patient_id text,    // e.g., P1234  
  slide_num int,      // e.g., 5 (out of 10, for this patient)  
  x int, // x coordinate within slide  
  y int, // y coordinate within slide  
  label int, // e.g., 0 = normal, 1 = tumoral  
  patch_id uuid,      // e.g., 6559dffd-e951-453c-9a46-ce ...  
  PRIMARY KEY ((patient_id, slide_num, label), x, y)  
);
```

- This table contains only the **minimum data** needed by the training
- Data retrieved via **efficient queries** to **single-row partitions**
- **More than one data table** may exist for the same dataset
- E.g., one might also save a **color-normalized** dataset, or both a **compressed** (JPEG) and **not compressed** (TIFF) one

```
CREATE TABLE patches.data_by_uuid (  
    patch_id uuid,  
    label int,  
    data blob, // image file (JPEG, TIFF, etc.)  
    PRIMARY KEY ((patch_id))  
);
```

```
from cassandra_dataset import CassandraDataset
from cassandra.auth import PlainTextAuthProvider
```

```
## Cassandra connection parameters
```

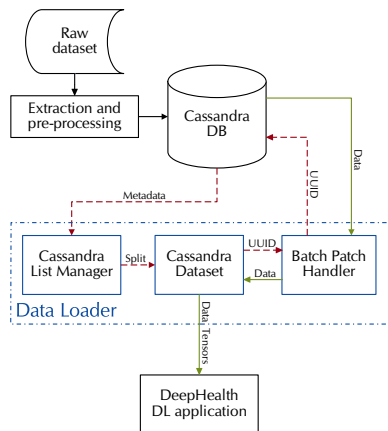
```
ap = PlainTextAuthProvider(username='user', password='pass')
```

```
cd = CassandraDataset(ap, ['cassandra-db'])
```

- This is the **main interface** for using the data loader
- It is written in **Python**
- It **transparently** calls the auxiliary classes

CassandraListManager: for managing the splits

BatchPatchHandler: for retrieving data and labels



- This **Python** class takes care of creating the splits, given the **desired target parameters**
- Images can be **aggregated** based on chosen keys
- Each aggregated partition (**group**) is assigned to a different split
- Once groups for each split have been computed, rows are extracted in round robin, to maximize diversity, until the **target values** are reached

```
## Create and save list of splits
cd.init_listmanager(
    table='patches.metadata_by_nat',
    id_col='patch_id',
    label_col="label",
    grouping_cols=["patient_id"],
    num_classes=2
)
cd.read_rows_from_db()
cd.init_datatable(
    table='patches.data_by_uuid'
)
cd.split_setup(
    split_ratios=[8,2],
    max_patches=100000
)
cd.save_splits(
    'splits/1M_3splits.pckl'
)
## One-line alternative:
# cd.load_splits(
#     'splits/1M_3splits.pckl'
# )
```

```
epochs = 50
split = 0 # training
cd.set_batchsize(32)
for _ in range(epochs):
    cd.rewind_splits(shuffle=True)
    for _ in range(cd.num_batches[split]):
        x,y = cd.load_batch(split)
        ## feed features and labels to DL engine [...]
```

- Low-level **C++** class
- Python bindings exposed via **pybind11**
- Data are read in **parallel** by a thread pool and **prefetched** in background, while the GPU is processing the previous mini-batch
- Data augmentations (via the **ECVL** library) are applied in background as well
- **Double-buffering** to reduce the DB + network latency
- Expensive system resources are allocated **lazily**

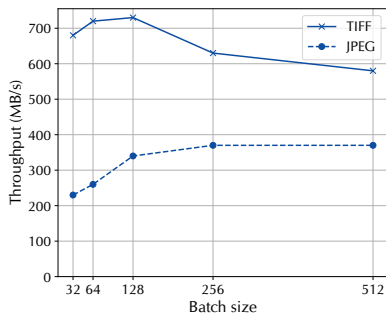
- Our data management strategy allows to **easily distribute** (and **uniformly, globally permute**) data enabling **distributed training**
 - Without need for a centralized process
- Consider a **parallel system** of **size n**
 - At startup, **each rank** reads the **full list of the images** hosted by the Cassandra servers (by querying the DB or by reading a pre-shared file)
 - The data loaders on each rank are initialized with the **same seed**
 - Each data loader creates **$2n$ splits**: n for **training** and n for **validation**
 - **Rank i** will read **training** data from split i and **validation** data from split $n + i$
 - At the **end of an epoch** the UUIDs in the training splits are **uniformly shuffled**, using the **same seed** on each rank

- Cluster of **18 nodes**
 - (Up to) **2 nodes**: running **Cassandra**
 - (Up to) **16 nodes**: **consuming** the data
- Intel Xeon E5-2680 v3 CPUs (**12 cores**, 2 threads/core)
- We are interested in the **data loading** stage
- 10 Gb/s **Ethernet**
- **Kubernetes** and **Docker** containerization
- **Portability** and **ease of deployment**

ImageNet-2017 dataset

- 166 GB
 - 1,281,167 images
 - 1000 classes
-
- We **resized** and **center-cropped** all the images to the standard resolution 224x224x3 (RGB)
 - We saved them as **BLOBs** in the Cassandra DB, both as
JPEG Compressed, quality: 90, average size: 20 kB
TIFF Not compressed, size: 150 kB
 - This **data preprocessing** step is easily parallelizable and **scalable** (no synchronizations are needed)
 - Parallelized with **PySpark**

- **Short-circuited** data loader: reads (and converts to tensor) as many images as possible, **without consuming the data**
- There are 32 parallel threads
- Larger batch sizes also benefit from **lower average latency**
- Up to a point in which **stress on the system** increases the DB retrieval latency
- CPU load: **1900%** at 18k images/s for **JPEG** and **400%** at 5k images/s for **TIFF**



GPU equivalent

A ResNet-50 consumes about 200 images/s. A **single data-loader** could thus sustain:

JPEG 50-90 GPUs

TIFF 19-24 GPUs

- The server is able to **saturate the bandwidth**, at about **1060 MB/s**, for both JPEG (20 KB images) and TIFF (150 KB images)
- Below we show Cassandra **retrieval latencies** at medium traffic and at network saturation
- Network saturation is obtained by using **16 data loaders**
- Latencies up to the 95th percentile remain **almost constant**
- The **99th percentile** grows approximately by a **factor 2**

GPU equivalent per Cassandra node

Each node could serve up to **250 GPUs (JPEG)** or **35 GPUs (TIFF)**

Percentile [%]	Time@MED [μs]	Time@SAT [μs]
50	88.15	88.15
95	152.32	152.32
99	263.21	454.83

JPEG (MED is about 370 MB/s)

Percentile [%]	Time@MED [μs]	Time@SAT [μs]
50	454.83	454.83
95	943.13	1131.75
99	1955.67	4055.27

TIFF (MED is about 630 MB/s)

Scaling up/down Cassandra DB

- Nodes can be added/removed **without any service disruption**
- When doing so the network bandwidth remains **saturated**

Performance of parallel filesystems

- Cutting-edge parallel filesystems achieve (raw performance)
latency of about 100 μ s
transaction rate of 250,000 operations/s per node
- By also exploiting **RDMA transfers**
- But they do not simplify the **management of splits and metadata**
- In our case, we hit **the network barrier at 50,000 transfers/s**
- We think that our design can be of particular interest for **small and medium size systems**, showing a **good trade-off** among performance, cost and ease of deployment

- **Free software** under the MIT License
- Available on **GitHub**
- <https://github.com/deephealthproject/CassandraDL>
- **Docker container** for easy testing, complete with Cassandra server and example datasets
- Detailed **paper**: <https://ieeexplore.ieee.org/document/9672005>



GitHub link

- **Free software** under the MIT License
- Available on **GitHub**
- <https://github.com/deephealthproject/CassandraDL>
- **Docker container** for easy testing, complete with Cassandra server and example datasets
- Detailed **paper**: <https://ieeexplore.ieee.org/document/9672005>



GitHub link

Thanks for your attention!