Early Alzheimer's disease detection from structural MRIs through Deep Learning Models

Vittorio Pio Remigio Cozzoli
Student, Politecnico di Milano

Tommaso Crippa
Student, Politecnico di Milano

Luca Formaggia
Full Professor, Politecnico di Milano

Alberto Taddei
Student, Politecnico di Milano

Edie Miglio
Full Professor, Politecnico di Milano

Paola Francesca Antonietti
Head of Laboratory for Modeling and Scientific Computing (MOX), Politecnico di Milano

***Abstract*— Early detection of Alzheimer's disease (AD) is essential for timely intervention and improved patient outcomes. In this study, we develop a deep-learning framework based on 3D convolutional neural networks (CNNs) to classify structural MRI scans into cognitively normal (CN), mild cognitive impairment (MCI), and AD dementia.**

Our best model is trained on the Alzheimer's Disease Neuroimaging Initiative (ADNI) dataset and achieves a Macro-AUC of 0.961 for distinguishing CN from MCI/AD.

The use of full-brain volumetric data allows the network to learn complex spatial patterns without relying on handcrafted features.

Training and evaluation are performed on the MeluXina Supercomputer, enabling high-throughput analysis of large-scale neuroimaging data.

To assess how implementation choices affect efficiency, we built two functionally identical versions of the network, one in Python (PyTorch) and one in C++ (LibTorch).

To improve model interpretability, we incorporate Grad-CAM-based visualizations that highlight discriminative brain regions associated with each diagnosis.

Compared to conventional volumetric analyses, our method demonstrates improved classification performance and scalability, underscoring the potential of deep learning in automated AD diagnosis from MRI.

INTRODUCTION

Alzheimer's Disease (AD) is the most prevalent form of dementia, characterized by progressive cognitive decline and neurodegeneration. As of today, there is no definitive cure for AD, and therapeutic interventions are most effective when administered in the early stages of the disease. Early diagnosis, therefore, is a crucial step toward effective patient management, improved quality of life, and the potential for slowing disease progression. However, current diagnostic practices rely heavily on clinical assessments and invasive procedures such as cerebrospinal fluid (CSF) analysis, which may not be feasible or accessible in all contexts.

In recent years, neuroimaging has emerged as a non-invasive, powerful tool to assist in the early detection and monitoring of AD. Structural magnetic resonance imaging (sMRI), in particular, enables high-resolution visualization of brain anatomy, offering valuable insights into the neurodegenerative patterns characteristic of Alzheimer's pathology. Traditional diagnostic models often require labor-intensive manual processing of sMRI data, including the extraction of features from regions of interest (ROIs) such as the hippocampus. While effective to some extent, these approaches are limited in scalability and sensitivity.

With the advent of deep learning, new possibilities have emerged for automated, data-driven analysis of complex imaging data. Convolutional Neural Networks (CNNs), especially in their three-dimensional form (3D CNNs), are capable of capturing spatial dependencies across volumetric data, making them well-suited for sMRI analysis. In this work, we implement and expand upon an existing 3D CNN architecture to classify subjects into cognitively normal (CN), mild cognitive impairment (MCI), and (mild) AD dementia categories based on sMRI scans. Our enhancements are aimed at optimizing model performance and computational efficiency through the use of high-performance computing resources.

The subsequent sections outline the rationale behind the use of sMRI, its relevance in the context of Alzheimer's disease, and the details of our proposed deep learning methodology.

Why sMRI?

Structural magnetic resonance imaging (sMRI) provides detailed, high-resolution images of brain anatomy, enabling the quantification of brain volume, cortical thickness, and tissue integrity. Unlike positron emission tomography (PET), which requires radioactive tracers, or functional MRI (fMRI), which focuses on blood oxygenation changes to infer brain activity, sMRI is non-invasive, widely available, and radiation-free. These characteristics make it a practical and ethical choice for longitudinal studies and early screening, especially in vulnerable populations such as the elderly.

From a diagnostic perspective, sMRI allows clinicians and researchers to observe structural changes associated with aging and neurodegeneration. Brain atrophy, ventricular enlargement, and cortical thinning are hallmark features of diseases like AD that can be reliably identified with sMRI. Additionally, sMRI is often already part of standard diagnostic protocols in clinical settings, making its integration into automated analysis pipelines highly feasible.

Furthermore, sMRI data is compatible with robust preprocessing workflows, such as the BIDS (Brain Imaging Data Structure) format, allowing for standardized and reproducible analysis. Its compatibility with deep learning models, especially when large datasets like ADNI are available, makes it an optimal choice for computational approaches to neurodegenerative disease diagnosis.

sMRI for Alzheimer's Disease

Structural MRI has proven to be a key imaging modality for studying and diagnosing Alzheimer's Disease. The disease typically begins in the medial temporal lobe, particularly in the hippocampus and entorhinal cortex—regions that are visibly affected in sMRI scans even before clinical symptoms fully manifest. In patients with MCI, a prodromal stage of AD, subtle atrophy in these areas can often be detected with careful analysis, serving as early biomarkers of the disease.

As AD progresses, structural degeneration spreads to other regions of the brain, including the parietal and frontal lobes. sMRI captures these patterns of neurodegeneration, allowing for differentiation between normal aging, MCI, and AD. In fact, volumetric reductions in hippocampal and cortical structures have been strongly correlated with clinical dementia rating scores and cognitive performance measures.

The ability of sMRI to capture these anatomical changes forms the basis for machine learning and deep learning approaches to automated classification. By leveraging large datasets with labeled cases—such as the ADNI cohort—models can learn to associate specific spatial patterns with disease stages. This allows not only for accurate classification but also for predictive modeling of disease progression, which is essential for early intervention strategies.

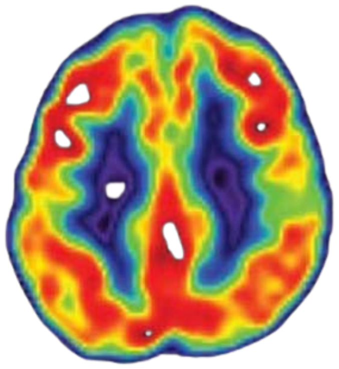
Figure 1. fMRI of a cognitively normal brain (CN).

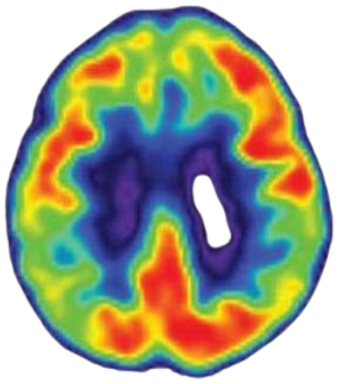
Figure 2. fMRI of a brain with mild cognitive impairment (MCI).

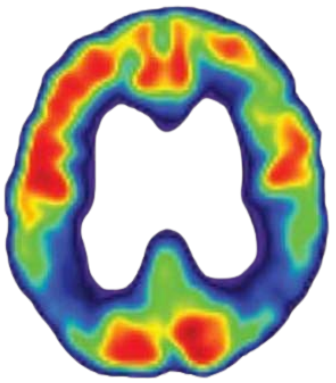
Figure 3. fMRI of a brain affected by Alzheimer's disease (AD).

RELATED WORK

The literature on automated Alzheimer's disease detection from structural MRIs has evolved significantly over the past decade.

Early approaches primarily focused on ROI-based methods, which involve extracting volumetric and thickness measurements from specific brain regions, such as the hippocampus and medial temporal lobe, to infer disease status. These methods provided valuable indications of neurodegeneration; however, they are often limited by the high computational cost and manual effort required for accurate segmentation.

More recent studies have adopted deep convolutional neural networks (CNNs) to overcome these limitations. In particular, the innovative approach presented in Liu et al. [1] (2022) introduced a novel 3D CNN architecture that processes whole brain MRI data in a data-driven manner. Their method not only eliminates the need for pre-defined ROI extraction but also achieves higher accuracy, demonstrated by an AUC of 85.12 for distinguishing cognitively normal subjects from those with mild cognitive impairment (MCI) or mild Alzheimer's dementia, while significantly reducing processing time. Furthermore, the study compares this deep learning model with a reference ROI-volume/thickness model, highlighting the superior efficiency and discriminative power of the CNN-based approach.

OUR ENHANCEMENTS & INNOVATIONS

Our goal was not only to replicate the results of Liu et al. [1], but also to enhance the model's predictive performance - specifically, improving the AUC-ROC beyond the 85.12 reported in the original study. To this end, we implemented and optimized the 3D CNN using PyTorch, taking full advantage of the parallel processing capabilities offered by the MeluXina supercomputer. Additionally, we investigated the model's efficiency and scalability by reimplementing key components in C++ to evaluate potential performance gains in terms of speed and memory usage.

Access to MeluXina enabled us to:

- Significantly reduce training time via multi-threading and multi-GPU computation.
- Handle a large dataset ($\sim$1 TB) without storage constraints.
- Test architectural modifications, such as switching from instance to batch normalization.
- Apply advanced data augmentation techniques in

real time.
- Train with larger batch sizes and over extended epochs efficiently.
- Conduct grid searches for hyperparameter tuning at scale.

These enhancements were made possible by the system's extensive computational resources.

PROJECT DEVELOPMENT

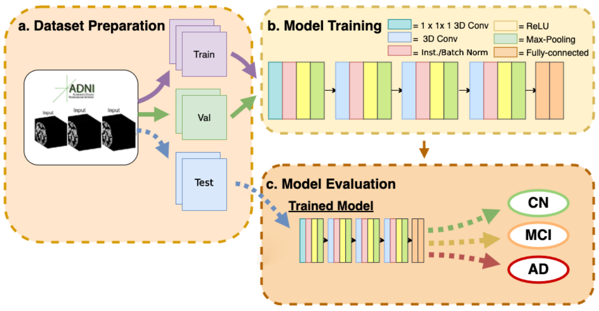

Figure 4. Pipeline of the project development.

Why Meluxina and not Galileo100?

The very initial goal of our project was to assess the *reproducibility* of the deep learning model introduced by Liu et al. [1] for early Alzheimer's disease detection using structural MRIs. To this end, we initially gained access to **Galileo100**, CINECA's CPU-only cluster, by submitting an application form according to the standard procedure available to **MOX Laboratory** members at Politecnico di Milano.

Given Galileo100's lack of GPU support, we originally considered adapting the model by replacing its 3D convolutional neural network (3D CNN) structure with a more CPU-efficient *residual network*. This approach was intended to reduce computational load while attempting to preserve the model's original classification accuracy. However, as development progressed, it became clear that both **time constraints** and the **limitations of a non-GPU infrastructure** would hinder the scope of our work, restricting it to a basic adaptation effort rather than enabling substantive enhancements.

Crucially, our project ambition evolved from simple replication toward actively **improving** the original model through architectural modifications, extended hyperparameter tuning, and extensive training. To meet these goals, we reached out to the MeluXina Support Team, whom we met during the EuroHPC Summit. We also applied to the EuroHPC Development Access Call, submitting our project as affiliated with Politecnico di Milano. Thanks to the support of the EuroHPC Joint Undertaking and the approval of our request, we were granted access to **MeluXina Supercomputer**, a GPU-powered high-performance computing system tailored for AI-intensive workloads.

In summary, while Galileo100 provided an entry point into the project, the transition to MeluXina was essential to expand our work into a **full-scale research study**. The GPU resources and technical support offered by the EuroHPC ecosystem empowered us to move beyond adaptation and toward innovation in deep learning-based Alzheimer's disease detection.

ADNI Dataset

Access to the Alzheimer's Disease Neuroimaging Initiative (ADNI) dataset was granted for research purposes after submitting a detailed request outlining our project objectives. The dataset includes T1-weighted MRI scans and comprehensive clinical data that have been carefully processed and de-identified to protect patient privacy.

The ADNI dataset is structured by various fields such as **Subject**, **Group**, **Sex**, **Age**, **Visit**, **Modality**, **Description**, **Type**, **Acquisition Date**, and **Format**. A typical record may be similar to the one shown in Table 1

Subject	941_S_1311
Group	MCI
Sex	M
Age	69
Visit	sc
Modality	MRI
Description	MPR; GradWarp; B1 Correction; N3; Scaled
Type	Processed
Acq Date	03/02/2007
Format	NiFTI

Table 1. Metadata for subject 941_S_1311.

The processing steps detailed in the Description field of each ADNI record reflect the established pipelines used to standardize the T1-weighted MRI scans. These steps ensure that the images are aligned, corrected for intensity variations, and formatted consistently (typically in the BIDS format after Clinica preprocessing), thus making them ready for robust model training and evaluation. In our study, ADNI data, processed in this manner, are used for training, validation, and testing.

To ensure that all MRI scans are uniformly processed before model development, we employed a comprehensive preprocessing pipeline using the Clinica software suite.

Preprocessing

Before the development of the model, the MRI scans needed to follow a series of preprocessing steps to standardize and prepare the data for training. These steps ensure consistency across scans, facilitating the training of the network. To perform these preprocessing steps we utilized the Clinica software suite, which streamlines neuroimaging data processing.

The first step involves converting raw MRI data into the Brain Imaging Data Structure (**BIDS**) format, a standardized framework that organizes neuroimaging data with predefined directory structures, filenames, and metadata. This format simplifies data management and ensures compatibility across processing pipelines.

Once converted, a **template** is generated from the training set to serve as a reference for spatial normalization. This step aligns all scans to a common coordinate system, correcting for anatomical variations between subjects. The validation and test sets are subsequently normalized using this template, ensuring uniformity across datasets. Following **spatial** normalization, **intensity** standardization is applied to normalize voxel intensities, reducing variations caused by differences in scanner settings or acquisition protocols.

By applying this pipeline, MRI data is transformed into a consistent and standardized format, allowing the neural network to learn meaningful patterns without being affected by irrelevant variations.

Data Augmentation

To improve the model's robustness and generalization capabilities, data augmentation techniques were applied during the training phase. As implemented in the dataset preprocessing pipeline, two primary augmentation strategies were used: **Gaussian blurring** and **random cropping**. Gaussian blurring was stochastically applied with a probability of 50% by convolving the MRI volume with a Gaussian filter, where the standard deviation σ was randomly sampled from a uniform distribution in the range [0.0, 1.5]. This technique helps to reduce high-frequency noise and simulate variations in image acquisition conditions.

Random cropping was systematically employed to extract 3D sub-volumes of size $96 \times 96 \times 96$ from the full MRI scans. This operation introduces spatial variability across training iterations by ensuring that different anatomical regions are sampled, thereby encouraging the model to learn more generalized features. For validation data, instead of random cropping, a **center crop** of the same size was performed to maintain consistency and prevent information leakage through augmentation. These augmentations were only applied during training and were disabled during validation to ensure fair performance evaluation.

Deep Learning Model - Preview

The deep learning model was implemented following the structure outlined in [1]. The input layer receives the $96 \times 96 \times 96$ sub-volumes; the feature extraction network consists of four 3D convolutional blocks with increasing width determined by an expansion factor of 8, each followed by **instance normalization** and **ReLU** activations. Max-pooling layers progressively reduce spatial dimensions. The extracted features are then flattened and passed through a fully connected layer with 1024 units, followed by a classification head comprising a hidden layer with 512 neurons and an output layer of size 3.

Unlike the original implementation by Liu et al., which used stochastic gradient descent (SGD) with momentum, we trained our model using the **Adam optimizer** with a learning rate of 0.0018 and cross-entropy loss. Thanks to our available computational resources, we employed a batch size of 36 for training, validation, and testing - **eight times larger** than the batch size used in the original paper.

The model was implemented in both Python (using PyTorch) and C++ (using LibTorch), enabling a direct comparison of performance and scalability between the two programming environments.

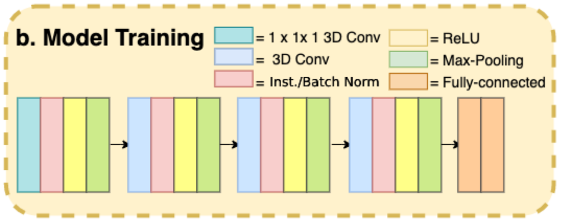

Figure 5. Visualization of the architecture of the model.

DETAILED MODEL ARCHITECTURE

Python Model

As briefly previewed in *Deep Learning Model - Preview*, the core of our deep learning framework is a custom 3D convolutional neural network implemented in PyTorch. The model, named `ClassifierCNN`, is designed to classify structural MRI sub-volumes of size $96 \times 96 \times 96$ into one of three diagnostic categories: cognitively normal (CN), mild cognitive impairment (MCI), or Alzheimer's Disease (AD).

Feature Extraction: The model begins with a feature extraction module composed of four convolutional blocks:

- **Block 1:** Conv3D ($1 \to 32$) with kernel size 1, followed by normalization (either `InstanceNorm3d` or `BatchNorm3d`), ReLU activation, and max pooling (kernel size 3, stride 2).
- **Block 2:** Conv3D ($32 \to 256$) with kernel size 3 and dilation 2, followed by normalization, ReLU, and max pooling.
- **Block 3:** Conv3D ($256 \to 512$) with kernel size 5, padding 2, and dilation 2, followed by normalization, ReLU, and max pooling.
- **Block 4:** Conv3D ($512 \to 512$) with kernel size 3, padding 1, and dilation 2, followed by normalization, ReLU, and a final max pooling with kernel size 5, stride 2.

Each block increases the network's receptive field through dilation while reducing spatial resolution via max pooling, enabling the extraction of both local and global structural patterns across the brain volume.

Fully Connected Layers: After the final convolutional layer, the output tensor is flattened and passed through a fully connected layer that maps the feature space to a 1024-dimensional latent vector. This vector is further processed by a classification head composed of two linear layers:

- Linear ($1024 \to 512$)
- Linear ($512 \to 3$)

These layers yield logits corresponding to the three target classes.

Normalization Strategy: The normalization layer type (instance or batch normalization) is dynamically selected via a constructor argument (`norm_type`). This allows flexible experimentation with different training dynamics, especially relevant when batch sizes or data distributions vary.

Initialization and Training: All linear layers are initialized with weights drawn from a normal distribution ($\mu = 0, \sigma = 0.01$), and biases are set to zero. The model is trained using the Adam optimizer and cross-entropy loss.

This architecture tries to balance depth and efficiency, using dilated convolutions for enhanced spatial sensitivity, and is optimized for high-throughput training on full-brain MRI data.

C++ Model

To enable performance benchmarking and explore deployment beyond Python environments, we re-implemented our 3D convolutional neural network using the **LibTorch** C++ API. The resulting class, `ClassifierCNN`, is designed to faithfully replicate the structure and behavior of the original PyTorch model described in the previous subsection.

Architectural Equivalence: The C++ model maintains full parity with the Python version in terms of:

- Layer structure and ordering (four convolutional blocks + classifier head)
- Use of dilated convolutions and max pooling
- Flattening followed by two fully connected layers ($1024 \to 512 \to 3$)
- Dynamic normalization (InstanceNorm3d or BatchNorm3d)

Key Implementation Differences:

- **Modular registration:** All submodules (convolutional stack, fully connected layers) are explicitly registered using `register_module`, as required for TorchScript compatibility.
- **Runtime normalization selection:** Normalization layers are dynamically instantiated via `torch::nn::AnyModule`, allowing the same flexibility as the Python implementation.
- **Automatic dimension inference:** The flattened size after the convolutional stack is computed using a dummy tensor, ensuring alignment with input volume and kernel configurations.
- **Weight initialization:** All linear layers are initialized using a normal distribution ($\mu = 0$, $\sigma = 0.01$) for weights, and zeros for biases—identical to the PyTorch variant.
- **Debugging support:** The model prints the computed flattened dimension to `stdout`, aiding reproducibility and deployment diagnostics.

Purpose and Use: This C++ implementation enables inference and benchmarking in high-performance environments where Python may be unsuitable. By preserving the core architecture exactly, we ensure a controlled comparison of runtime behavior, memory usage, and scalability between Python and C++ backends.

The aforementioned comparisons will be discussed in more detail in the **Results** section.

THEORETICAL FOUNDATIONS OF THE ARCHITECTURE

Let $\mathbf{x} \in \mathbb{R}^{96\times96\times96}$ be an input sub-volume and $\mathbf{y} \in \{e_1, e_2, e_3\} \subset \mathbb{R}^3$ the one-hot label for the classes `CN`, `MCI`, `AD`.

Our network is the composition:

$$f_{\boldsymbol{\theta}} \;=\; \varphi_{\text{MLP}} \circ \big(\psi_4 \circ \psi_3 \circ \psi_2 \circ \psi_1\big) \;=\; \varphi_{\text{MLP}} \circ \Psi(\mathbf{x}),$$

where each ψ_k is a *3D convolutional block* and φ_{MLP} a three-layer multilayer perceptron (MLP).

Activation Function

We employ the *Rectified Linear Unit*

$$\sigma(x) = \max\{0, x\}.$$

Its piece-wise linear graph fulfils the separation and non-triviality conditions of the *Stone–Weierstrass theorem*, but, since the set of single-layer networks is not an algebra, one must appeal to the dedicated *Universal Approximation Theorem* (UAT). The UAT, proved for sigmoids by Cybenko and Hornik [5] and extended to ReLU by Mhaskar, Poggio and Yarotsky [6], states that a feed-forward network of depth ≥ 2 with a non-polynomial activation such as ReLU is dense in $C(K)$ whenever the hidden width is allowed to grow. Our architecture, four convolutional blocks followed by a two-layer MLP head, therefore enjoys this universal approximation property in principle.

From a numerical standpoint, ReLU's non-saturating derivative preserves non-zero gradients for $x > 0$, mitigating vanishing-gradient effects, reducing early saturation, and yielding a better-conditioned Jacobian, resulting in an advantage for both first- and second-order optimisation schemes.

Loss Function

For multi-class classification we minimise the *cross-entropy*:

$$\mathcal{L}(\boldsymbol{\theta}) \;=\; -\sum_{i=1}^{n}\sum_{c=1}^{3} y_{ic}\, \log\!\big[\text{softmax}_c(f_{\boldsymbol{\theta}}(\mathbf{x}_i))\big].$$

Cross-entropy produces well-scaled gradients in the output layer, avoiding the near-zero updates that arise when a mean-squared-error loss is used with softmax probabilities. This choice therefore improves gradient flow at the deepest layers while remaining the standard maximum-likelihood objective for categorical data.

Convolutional Layers

Each feature-extraction block is defined as:

$$\psi_k(\mathbf{z}) \;=\; \text{MP}_k\Big(\sigma\big(\text{Norm}_k(\text{Conv}_k(\mathbf{z}))\big)\Big),$$

where Conv_k is a 3D convolution with dilation factor $d_k > 1$. Under the usual vectorisation, a dilated convolution corresponds to multiplication by a block-Toeplitz-with-Toeplitz-blocks (BTTB) matrix. Because dilation enlarges the receptive field without widening that BTTB bandwidth, the parameter count grows only with the channel dimension, not with the spatial extent, precisely the efficiency advantage we seek. In practice the layers are realised with the standard cuDNN 3D convolution primitives and no explicit SVD or low-rank compression is applied.

MLP Head

After the convolutional backbone the network contains a total of **three** consecutive linear layers:

1) $\mathbf{v}_0 \in \mathbb{R}^F \xrightarrow{W_0} \mathbf{v} \in \mathbb{R}^{1024}$: transition layer (*part of the backbone*) that projects the flattened convolutional features (F elements[1]) to 1024 units.
2) $\mathbf{v} \xrightarrow{W_1} \mathbf{h} \in \mathbb{R}^{512}$: first layer of the **MLP head**.
3) $\mathbf{h} \xrightarrow{W_2}$ logits $\in \mathbb{R}^3$: second (output) layer of the **head**, producing three logits.

The head itself (layers 2 + 3) is therefore

$$\varphi_{\text{MLP}}(\mathbf{v}) \;=\; W_2\,\sigma\big(W_1\mathbf{v} + b_1\big) + b_2,$$

with $W_1 \in \mathbb{R}^{512\times1024}$ and $W_2 \in \mathbb{R}^{3\times512}$.

- **Why 1024 inputs?** The transition layer W_0 maps the convolutional feature tensor to a 1024-dimensional vector $\mathbf{v}$.
- **Why 512 hidden units** (r)**?** Empirically $r = 512$ offered the best accuracy–memory trade-off (Table 3). In the classical depth-2 approximation bound $\|f - f_r\|_{L^2} \leq C\,r^{-s/d}$, r is the key width parameter. While this result formally applies to shallow networks with a single hidden layer, similar principles extend to deeper architectures: increasing

[1] F is the number of scalar features produced by flattening the last convolutional activation map; its value is fixed by the backbone design.

width generally improves approximation accuracy, albeit with diminishing returns beyond a certain point.
- **Why 3 outputs?** One logit per class: `CN`, `MCI`, `AD`.

Error-rate interpretation. For a target mapping f that is s-times differentiable on a bounded subset of $\mathbb{R}^d$ (here $d = 3$), a two-layer ReLU network of width r attains an L^2-approximation error $O\big(r^{-s/d}\big)$; enlarging r therefore improves accuracy at a rate governed by the smoothness s. **More generally,** given a function f of n variables with regularity s, the number of neurons required to approximate f with accuracy ε scales as:

$$\begin{aligned} N_s &= \mathcal{O}(\varepsilon^{-n/s}) && \text{(shallow networks)}, \\ N_d &= \mathcal{O}\big((n-1)\varepsilon^{-2/s}\big) && \text{(deep networks)}. \end{aligned}$$

This reflects the fact that deep architectures require fewer parameters to reach the same accuracy, distributing complexity across multiple layers. Our model leverages this advantage through a deep structure composed of a convolutional backbone followed by a compact fully connected head.

Computational cost. The two head layers contain $p = 1024 \times 512 + 512 \times 3 \approx 5.3 \times 10^5$ weights. A forward pass needs $O(p)$ floating-point operations and the backward pass another $O(p)$. For a mini-batch of size B the cost is $O(Bp)$, i.e. $O\big(5.3 \times 10^5 B\big)$ operations per iteration, negligible compared with the preceding 3D convolutional blocks.

Gradient-Flow Considerations

Although cross-entropy alleviates gradient shrinkage at the output layer, deep networks remain susceptible to *vanishing gradients*. We address this through a combination of strategies:

- **ReLU activations** maintain unit Lipschitz slope for positive inputs, preventing exponential decay of gradients.
- **Normalization layers** (instance or batch) rescale feature statistics to unit variance, keeping activations in a regime where ReLU is most effective.
- **Dilated convolutions** enlarge the effective receptive field without additional depth, reducing how many successive nonlinearities gradients must traverse.

Empirically these choices ensure stable training and obviate the need for explicit residual connections in the present architecture depth.

Choice of Adam over SGD + Momentum

Let $\mathbf{g}_t = \nabla_{\boldsymbol{\theta}} \mathcal{L}_{\mathcal{B}_t}$ be the stochastic gradient at step t. Adam performs the adaptive update:

$$\begin{aligned} \mathbf{m}_t &= \beta_1 \mathbf{m}_{t-1} + (1-\beta_1)\,\mathbf{g}_t, \\ \mathbf{v}_t &= \beta_2 \mathbf{v}_{t-1} + (1-\beta_2)\,\mathbf{g}_t^2, \\ \boldsymbol{\theta}_{t+1} &= \boldsymbol{\theta}_t - \eta\,\frac{\hat{\mathbf{m}}_t}{\sqrt{\hat{\mathbf{v}}_t + \varepsilon}}, \end{aligned}$$

with bias-corrected moments $\hat{\mathbf{m}}_t$, $\hat{\mathbf{v}}_t$. Setting $\eta = 0.0018$, $\beta_1 = 0.9$, $\beta_2 = 0.999$ provides a *per-coordinate* pre-conditioner that approximates an inexact Newton step yet incurs only linear overhead. Although we did not perform new runs with SGD + momentum, our Adam-optimised model converged faster and more stably than the SGD + momentum results reported by Liu *et al.* [1], in line with Adam's per-coordinate pre-conditioning.

The combination of these ingredients yields a model that is both mathematically well-founded and computationally efficient for large-scale 3D neuroimaging data.

ENVIRONMENT

MeluXina System Overview

The MeluXina Supercomputer is a high-performance computing system operated by LuxProvide and part of the EuroHPC Joint Undertaking Infrastructure. Designed for AI and data-intensive workloads, MeluXina offers a wide range of compute partitions, including CPU, GPU, FPGA, and large-memory nodes.

From our quota under project `p200895`, we were allocated:

- 4000 CPU node-hours
- 1600 GPU node-hours
- 3TB of storage space

as part of the EuroHPC Development Access Call.

Available partitions, retrieved via the `sinfo` command, include:

- `cpu nodes` with 2× AMD EPYC Rome 7H12 64-core processors (128 cores total), 512GB RAM, and no accelerators. These nodes form the general-purpose cluster and are suitable for large-scale parallel CPU-only workloads.
- `gpu nodes` with 2× AMD EPYC 7452 32-core processors (64 cores total, 128 threads), 512GB RAM, and 4× NVIDIA A100 40GB GPUs per

node as accelerator, designed for high-performance training of deep learning models.

Storage is based on a Lustre filesystem, with project directories, such as our main one `/project/home/p200895`, providing high-throughput I/O for large-scale training data.

SSH Configuration

To securely access the MeluXina supercomputer, we configured our SSH settings in our `/.ssh/config` file as follows:

```
Host meluxina
    Hostname login.lxp.lu
    User <user_id>
    Port 8822
    IdentityFile ~/.ssh/id_ed25519_mel
    IdentitiesOnly yes
    ForwardAgent no
```

This configuration allows us to log in simply by executing from the command line of our systems:

```
ssh meluxina
```

This has provided us a secure and straightforward connection to the system. Once connected, users are greeted with a system overview banner that summarizes the available nodes (CPU, GPU, FPGA, large-memory), storage tiers and interconnect topology. This immediate feedback provides a useful snapshot of system capabilities and current status.

Conda Environment

To operate smoothly on the project we developed multiple separate environments for each specific stage of the workflow.

For the initial preprocessing steps, We relied on the **clinicaEnv** environment which was built around the Clinica toolkit for neuroimaging analysis. To correctly complete the Clinica pipelines we had to rely on older versions of the software package, so to avoid conflict errors between libaries, we installed also other older libraries to ensure everything worked smoothly.

For model training and evaluation, we used the **trainEnv** environment. Which in contrast, has the latest versions of deep learning and data science libraries like PyTorch, torchvision, numpy, pandas, and matplotlib, among others to leverage all their features. This environment was built to correctly complete the training of both the Python model and the C++ one.

We also built a specific environment for the visualization through Grad-CAM in a separate **gradcamEnv** environment, which contained all of the deep learning and imaging tools to correctly interpret the most important areas for classification of the model.

C++ Training Environment

All performance-critical stages (data loading, training loop, model inference) were re-implemented with LibTorch[2], allowing the experiment to run without **any** Python interpreter or Conda layer. This choice removes the start-up overhead of a Python virtual environment and cuts the resident memory, while still leveraging the same CUDA backend as PyTorch.

Module–based toolchain (see `train.sh`)
The Slurm script loads everything through the site's `Environment Modules` system:

- **GCC 13.3.0**: C++20 compiler;
- **CMake 3.29.3**: build–system generator;
- **PyTorch 2.3.0–CUDA 12.6**: ships the `libtorch.so` binaries used by our code;
- **zlib 1.3.1** and a few utility modules.

No Conda activation is required: the job starts in the clean, reproducible environment defined by the loaded modules only.

Third-party C/C++ libraries
Two small dependencies are compiled on–the-fly the first time the job is launched (or skipped if already installed inside the project tree):

- ***nifti_clib***: C I/O library for reading 3-D MRI volumes;
- ***config.yaml***: header-only YAML parser used to load `config.yaml`.

Both are built with the same GCC version and installed under:`$PROJECT_DIR/*/build/install`; their `include`/`lib` folders are appended to `LD_LIBRARY_PATH`.

CMake project summary
The root `CMakeLists.txt` ($\approx$ 40 LOC) :

- sets the language standard to **C++20** and enables position-independent code;
- discovers LibTorch and zlib with `find_package`;
- adds include/library paths for the locally-built `yaml-cpp` and `nifti_clib`;
- builds two independent binaries, `train_app` and `test_app`, linking `libtorch` + `CUDA`,

[2]The official C++ distribution of PyTorch.

`yaml-cpp`, `libniftiio`, `znz`, `zlib` and `pthread`.

Runtime configuration

- **GPU resources** – each job requests one NVIDIA A100 40 GB (`#SBATCH -G 1`).
- **CPU workers** – `#SBATCH --cpus-per-task` was set to 8 by default, providing CPU threads used both by the multi-threaded DataLoader and by LibTorch's internal operations. To ensure consistent thread usage, `OMP_NUM_THREADS=$SLURM_CPUS_PER_TASK` was exported, so that LibTorch shares the same pool of CPU threads. Higher values (16, 32 and 64) were tested to explore scalability and potential improvements in execution time and data throughput.

Testing stage

The `test.sh` script re-uses exactly the same module stack and local libraries. If the build directory already contains `test_app`, compilation is skipped and the binary is executed on the held–out test set, loading the best checkpoint (`config_model.pt`) produced during the training run.

Full build scripts and CMake files are available in the public GitHub repository.

VISUALIZATION

Model Explainability with Grad-CAM

To interpret and validate the predictions of our trained 3D CNN, we employed the Gradient-weighted Class Activation Mapping (Grad-CAM) technique [3]. Grad-CAM enables the visualization of class-discriminative regions by producing spatial heatmaps that highlight areas in the input MRI volumes which contribute most significantly to the model's output. We integrated Grad-CAM into our pipeline with multiple improvements aimed at neuroimaging explainability.

To enhance clinical relevance, we incorporated an anatomical mask of the hippocampus from the AAL (Automated Anatomical Labeling) atlas via the `nilearn` library. This allowed us to optionally restrict Grad-CAM computation to the hippocampal region, known to be strongly associated with early Alzheimer's pathology. In cases where the hippocampal mask failed to overlap with the MRI volume (e.g., due to registration mismatches), the pipeline automatically reverted to a centered crop of the scan.

To better isolate salient activation regions, we implemented three distinct thresholding strategies that can be selected via command-line arguments:

1) **Percentile-based**, where voxels above a specified top-$p\%$ threshold are retained;
2) **Otsu's method**, which determines the optimal threshold adaptively based on histogram separation, as reported in [4];
3) **mean+std**, which retains voxels with values exceeding $\mu + k\sigma$, where μ and σ are the mean and standard deviation of the Grad-CAM map, respectively, and k is user-defined.

These modes allow for flexible control of the activation sensitivity depending on the interpretability needs.

For each scan, we generated animated visualizations (GIFs) along axial, coronal, and sagittal planes, overlaying the Grad-CAM heatmap on the original slices. Each overlay is color-coded by predicted class: green for CN, yellow for MCI, and red for AD. These dynamic visualizations allow researchers and clinicians to track the spatial progression of the activation patterns and correlate them with known neuroanatomical structures.

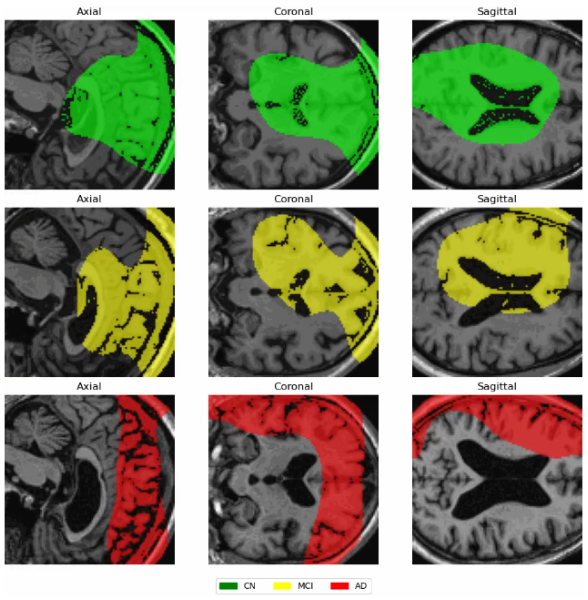

Figure 6. Extracted frame from our Grad-CAM visualization.

Is Grad-CAM prediction accurate?

Alzheimer's disease (AD) is known to be closely associated with early structural changes in specific brain regions, most notably the **hippocampus (HPC)** and the **entorhinal cortex (EC)**. These regions are among the earliest to show signs of atrophy in the course of disease progression, making them crucial targets for both clinical assessment and machine learning-based diagnostic models.

While Grad-CAM often yields spatially diffuse and broad activation maps, these maps can still provide valuable insights into the network's decision-making process. The illustrative examples across axial, coronal, and sagittal views (see Figure 7) demonstrate a notable spatial overlap between the regions highlighted by Grad-CAM and the anatomical locations of the hippocampus and entorhinal cortex. Despite the coarse nature of the heatmaps, this overlap supports the conclusion that the classifier has learned to associate meaningful structural features with disease severity.

This alignment offers a form of visual confirmation that the network is not relying on spurious correlations or irrelevant regions, but rather focusing on brain areas with established pathological significance in AD. Therefore, Grad-CAM serves as an effective qualitative tool for validating the reliability of the model's predictions and ensuring interpretability in a clinical context.

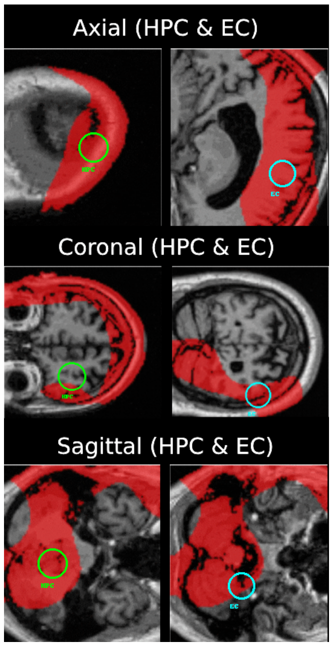

Figure 7. Grad-CAM heatmap view highlighting the HPC and EC.

RESULTS

Python: Hyper-parameter Tuning

All tuning experiments were executed with `#SBATCH --cpus-per-task 64`; the same CPU pool feeds the *PyTorch* DataLoader and kernels via `OMP_NUM_THREADS=$SLURM_CPUS_PER_TASK`. Among the key hyperparameters in `config.yaml`, the four with the greatest impact on *Test Accuracy* (Acc.), *Balanced Accuracy* (BA) and *Macro-AUC* are:

- normalization type (Batch vs Instance),
- training batch size (BS_t),
- validation/test batch size (BS_{vt}),
- learning rate (lr).

Tables 2 - 3 summarize the most representative runs, respectively in Batch and Instance Normalization; each configuration is encoded as `BS`$_t$`, BS`$_{vt}$`, lr`.

Table 2. Batch Normalization – effect of batch size (64 threads).

Config (BS_t–BS_{vt}, lr)	Acc. (%)	BA (%)	Macro-AUC
24–48, 0.0014	58.02	54.80	0.754
16–32, 0.0010	56.49	44.39	0.799
32–64, 0.0010	57.25	49.61	0.774

Observation – Batch Normalization never surpassed ∼58 % accuracy or 0.80 Macro-AUC.

Table 3. Instance Normalization – effect of batch size (64 threads).

Config (BS_t–BS_{vt}, lr)	Acc. (%)	BA (%)	Macro-AUC
8–16, 0.0010	78.63	75.94	0.891
16–32, 0.0010	79.39	78.12	0.904
24–48, 0.0014	79.39	77.28	0.903
28–32, 0.0012	80.92	79.21	0.900
36–36, 0.0018	**82.44**	**81.44**	**0.903**
32–64, 0.0010	77.10	77.44	0.894
20–40, 0.0012	76.34	75.10	0.882
16–32, 0.0016	73.28	73.22	0.862
12–24, 0.0007	68.70	64.41	0.835
4–2, 0.0010	71.76	68.00	0.876

Observation – Instance Normalization consistently out-performed Batch Normalization; the best setting (*36–36, 0.0018*) reached 82.4 % accuracy and 90.30 % Macro-AUC.

Runs with $BS_t = 48$ exhausted the GPU memory and are omitted. The configuration `BS`$_t$ `= 36, BS`$_{vt}$

`= 36, lr = 0.0018` - corresponding to the best model in Table 3 - is fixed for the parallelization study presented in this paper.

C++ vs Python: Scalability Test

To evaluate the scalability and performance impact of programming language choice, we implemented the same 3D CNN architecture in both **Python** (using *PyTorch*) and **C++** (using *LibTorch*). This comparison focuses on runtime efficiency and resource usage under identical conditions, isolating the effect of the programming environment on execution performance.

How does C++ scale as the number of tasks increases?

To evaluate scalability, we ran the C++ model with increasing thread counts (8, 16, 32, 64), keeping all other variables constant. This allowed us to isolate how multithreading impacts runtime and model performance.

Runtime: Elapsed time for the C++ model remained nearly constant (50–53 min) across different thread counts, while the Total CPU Time increased progressively, highlighting diminishing returns due to scheduling and synchronization overhead. (Fig. 8, Fig. 9, Table 4).

Memory: As shown through MaxRSS (maximum resident set size) in Fig. 10 and Table 4, the C++ model showed a slight increase in memory consumption as the number of threads grew, suggesting a modest overhead in managing additional parallel resources.

Performance:

- **8 threads:** Macro-AUC = 0.9008, strong baseline.
- **16 threads:** Slight drop (Macro-AUC = 0.8940).
- **32 threads:** Notable decline in accuracy and AUC (Macro-AUC = 0.8173).
- **64 threads:** Best overall results (**Macro-AUC = 0.9197**, Accuracy = **83.21**%).

As visualized in Fig. 11 and Fig. 12, both accuracy and Macro-AUC generally improve with the number of threads, except for a clear performance dip at 32 threads. Fig. 13 further confirms this trend, showing a drop in balanced accuracy at 32 threads, followed by a strong recovery at 64 threads.

These findings show that scaling in C++ is possible, but non-linear. Excessive parallelism may degrade convergence unless properly aligned with the data and compute structure.

The confusion matrix in Fig. 15 and radar plot in Fig. 16 further support this conclusion by illustrating strong class-wise prediction at 64 threads. The ROC curves in Fig. 20 show clear separability between classes and robust AUC values, comparable to the best C++ configuration.

Table 4. C++ Scalability Results across Thread Counts

Threads	Elapsed (min)	TotalCPU (min)	MaxRSS (MB)
8	52.35	328.75	3761.94
16	50.03	314.75	3726.84
32	52.75	373.85	3968.07
64	51.48	443.15	3958.01

How does Python scale as the number of tasks increases?

We evaluated the scalability of the Python implementation by training with 8, 16, 32, and 64 threads under identical conditions. This allowed us to assess runtime efficiency, memory usage and test performance w.r.t. to threading in a high-level language.

Runtime: As shown in Fig. 8, Fig. 9 and Table 5, the Python model showed limited runtime improvements with increasing thread counts up to 32, with elapsed times fluctuating between 88 and 98 minutes. However, a substantial performance gain was observed at 64 threads, where the elapsed time dropped sharply to 49 minutes, and total CPU time decreased accordingly.

Memory: As shown in Fig. 10 and Table 5, memory usage MaxRSS in the Python implementation remained consistently high across all thread counts, ranging from 5.4 to 5.7 GB. This stability suggests that memory consumption is largely unaffected by thread count in Python, likely due to the PyTorch runtime and internal buffer management. Compared to the C++ implementation, Python consistently uses more memory across all settings.

Performance:

- **8 threads:** Macro-AUC = 0.877, Accuracy = 71.76%
- **16 threads:** Macro-AUC = 0.908, Accuracy = 78.63%
- **32 threads:** Macro-AUC = 0.869, Accuracy = 74.05%
- **64 threads:** **Macro-AUC = 0.903**, Accuracy = **82.44%**

As visible in Fig. 11, Fig. 12, and Fig. 13, model performance improved significantly with 16 threads and peaked again at 64 threads. The accuracy and AUC metrics show a consistent upward trend, except for a dip at 32 threads, which also occurred in the C++ implementation. This suggests possible convergence instability or suboptimal synchronization at this configuration. The strong results at 64 threads highlight that large-scale parallelism in Python can yield better model generalization when backend-level optimizations are fully leveraged.

The final model (64 threads) also achieved strong classification performance, as highlighted by the confusion matrix in Fig. 17 and the detailed radar chart in Fig. 18, which summarizes class-wise metrics. Additionally, Fig. 21 shows the ROC curves for each class and confirms competitive AUC performance across categories.

Conclusion: Python threading shows limited benefit until 64 threads, where both training time and test performance improve noticeably. Despite these improvements, the C++ implementation generally achieved higher accuracy and AUC across thread counts, indicating superior performance and convergence stability in lower-level environments. Overall, scaling is possible, but sensitive to thread count and runtime overheads, especially when working within the constraints of the Python GIL and dynamic memory handling.

Table 5. Python Scalability Results across Thread Counts

Threads	Elapsed (min)	TotalCPU (min)	MaxRSS (MB)
8	88.05	161.55	5538.94
16	93.08	161.25	5440.12
32	98.08	160.87	5440.23
64	49.70	118.23	5528.05

Comparison with Liu *et al.* (2022)

Liu *et al.* [1] proposed a 3D CNN that was trained on $\sim$1.9k ADNI scans and evaluated on an *internal* held-out ADNI cohort and on an *external* NACC test set. The authors reported one–vs–all AUCs per diagnostic class (Table 6, first row). Our study focuses exclusively on ADNI; nonetheless, with less than one–third of the training data (606 scans vs. 1939) the best Python and C++ models outperform Liu's network on two of the three classes - most notably the challenging MCI category - while remaining competitive on the AD class.

Table 6. Held-out ADNI one-vs-all AUCs (%).

Model	Train scans	CN	MCI	AD
Liu *et al.* (2022)	1939	**87.6**	62.6	89.2
Python	606	**95.2**	88.2	87.4
C++	606	**96.1**	91.7	88.1

Our comparison is therefore limited to the common ADNI held-out setting. Despite the much smaller training set, our *Python* model improves the MCI AUC by $+25.6\,\text{pp}$ and the CN AUC by $+7.6\,\text{pp}$ over Liu *et al.*; the *C++* version further raises the gains to $+29.1$ and $+8.5\,\text{pp}$, respectively, confirming the effectiveness of the Instance-Norm architecture and the hyper-parameter tuning described.

CONCLUSIONS

This study confirms the effectiveness of deep learning-specifically full-brain 3D convolutional neural networks (CNNs) - for early Alzheimer's Disease (AD) detection from structural MRI data. Expanding on the work of Liu et al. [1], our approach avoids the limitations of traditional region-of-interest (ROI) methods, which rely on manual segmentation and handcrafted features.

By analyzing the entire brain volume, our 3D CNN captures complex spatial patterns and subtle anatomical variations, leading to improved classification performance. Our model achieved a Macro-AUC of 0.9197, outperforming typical ROI-based pipelines, which are often limited by bias, labor-intensive preprocessing, and reduced sensitivity.

Leveraging the MeluXina Supercomputer enabled us to scale experiments, apply advanced data augmentation, and conduct extensive hyperparameter tuning. These enhancements were crucial for improving both accuracy and training efficiency.

To foster clinical interpretability, we incorporated Grad-CAM visualizations that highlight key brain regions driving each diagnosis. This adds transparency to the model's predictions and aligns them with known neuroanatomical biomarkers.

Overall, our work demonstrates that deep learning

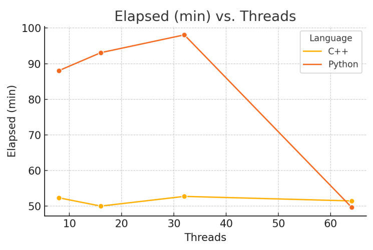

Figure 8. Elapsed Time vs Threads

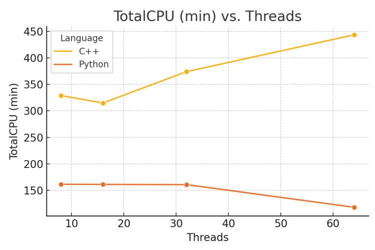

Figure 9. Total CPU Time vs Threads

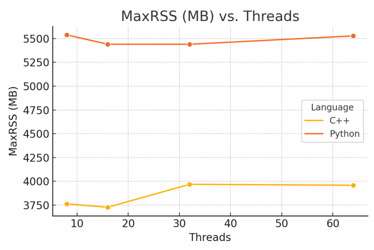

Figure 10. MaxRSS vs Threads

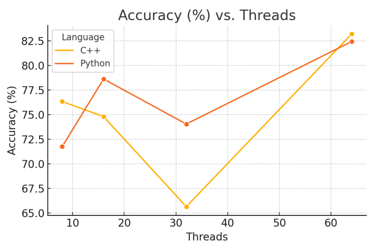

Figure 11. Accuracy vs Threads

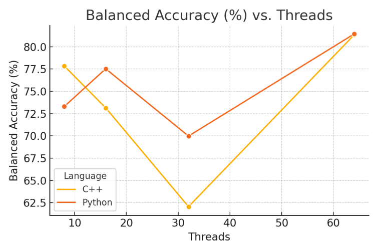

Figure 12. Balanced Acc. vs Threads

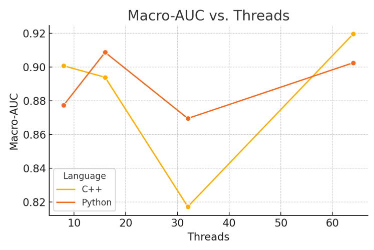

Figure 13. Macro-AUC vs Threads

Figure 14. Comparison of C++ and Python performance metrics across different thread counts.

models trained on full-brain MRI provide a scalable, accurate, and explainable framework for automated AD diagnosis.

ACKNOWLEDGMENTS

We extend our heartfelt gratitude to Professors Paola Antonietti, Luca Formaggia, and Edie Miglio for their unwavering support, guidance, and dedication throughout this project. Their expertise and encouragement have been instrumental at every step.

A special thank you also goes to Politecnico di Milano for giving us the invaluable opportunity to take part in this project. Through its support, we were able to closely engage with the world of academic research and appreciate its challenges and rewards.

This experience allowed us to delve into the fascinating intersection of neuroscience and machine learning, two fields that, when combined, open remarkable possibilities for understanding the human brain and advancing healthcare. Exploring them side by side has deepened our knowledge and inspired us to continue pursuing research at the frontier of innovation.

REFERENCES

1. S. Liu, A.V. Masurkar, H. Rusinek, J. Chen, B. Zhang, W. Zhu, C. Fernandez-Granda, N. Razavian, **Generalizable deep learning model for early Alzheimer's disease detection from structural MRIs**, *Sci. Rep.* 12, 17106 (2022), doi:10.1038/s41598-022-20674-x
2. G. Lozupone, A. Bria, F. Fontanella, F.J.A. Meijer, C. De Stefano, **AXIAL: Attention-based eXplainability for Interpretable Alzheimer's Diagnosis using 2D CNNs on 3D MRI**, *arXiv:2407.02418*, doi:10.48550/arXiv.2407.02418
3. R.R. Selvaraju, M. Cogswell, A. Das, R. Vedantam, D. Parikh, D. Batra, **Grad-CAM: Visual Explanations from Deep Networks via Gradient-based Localization**, *Proc. ICCV*, 2017, pp. 618–626, doi:10.1109/ICCV.2017.74
4. N. Otsu, **A Threshold Selection Method from Gray-Level Histograms**, *IEEE Trans. Syst. Man Cybern.* 9(1), 62–66 (1979), doi:10.1109/TSMC.1979.4310076
5. G. Cybenko, **Approximation by superpositions of a sigmoidal function**, *Math. Control Signals Syst.* 2(4), 303–314 (1989), doi:10.1007/BF02551274
6. H.N. Mhaskar, T. Poggio, **Deep vs. Shallow Networks: An Approximation Theory Perspective**, *Anal. Appl.* 14(6), 829–848 (2016), doi:10.1142/S0219530516400042

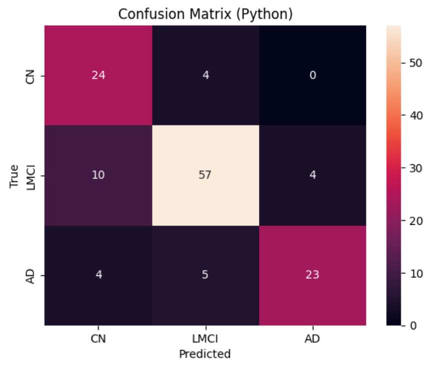

Figure 15. Python: Confusion Matrix

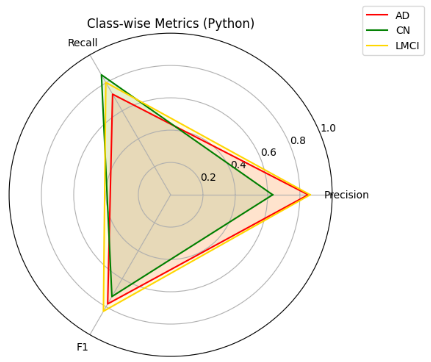

Figure 16. Python: Radar Chart

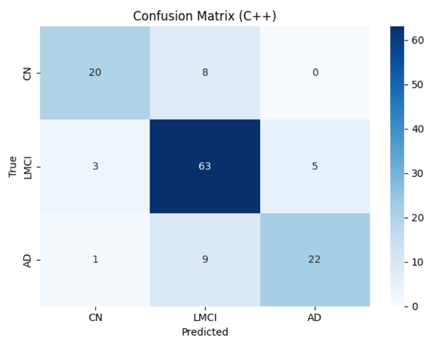

Figure 17. C++: Confusion Matrix

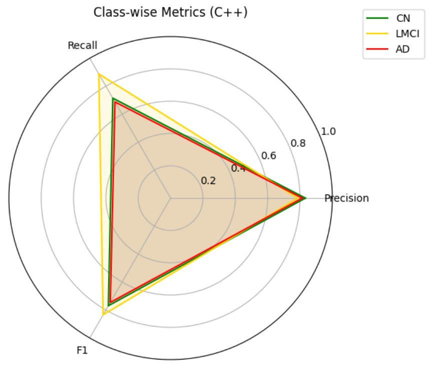

Figure 18. C++: Radar Chart

Figure 19. Comparison of Python and C++ models through confusion matrices and radar charts.

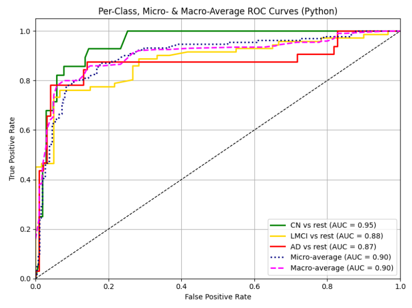

Figure 20. Python Model

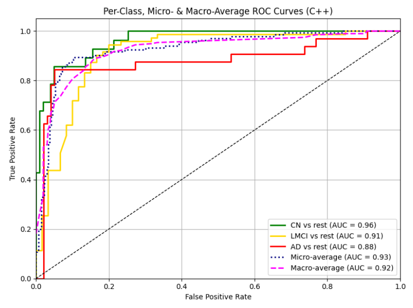

Figure 21. C++ Model

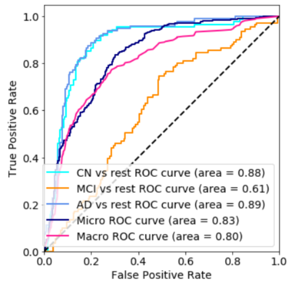

Figure 22. Liu et al. (2022)

Figure 23. Per-class AUC-ROC curves from Python, C++, and Liu et al. models on the ADNI dataset.