# Master 2 Computer Science

# Advanced Neural Networks Architectures

# Useful Tools

---

Akka Zemmari

Introduction

Introduction to GPUs

or how to speed up deep learning

Introduction to Dataloaders

or how to load and preprocess data

Overview of Keras, PyTorch, and PyTorch Lightning

or how to choose the right framework for your DL project

# Introduction

# Discleamer

This chapter aims to introduce some useful tools that we will use throughout this course.

The content serves as a brief overview rather than a comprehensive course, providing only the essential elements.

Each concept mentioned could warrant several chapters for a thorough exploration, but the emphasis is on their practical use.

# Introduction to GPUs

## or how to speed up deep learning

# What is a GPU, and why is it essential for deep learning?

## What is a GPU?

A Graphics Processing Unit (GPU) is a specialized processor originally designed to accelerate graphics rendering, primarily for video games and other visual applications.

Unlike a Central Processing Unit (CPU), which is designed to handle a wide range of tasks sequentially, a GPU is optimized for performing many calculations simultaneously, making it highly effective for parallel processing.

# What is a GPU, and why is it essential for deep learning?

## What is a GPU?

- Architecture: GPUs consist of thousands of smaller, efficient cores designed to handle multiple tasks simultaneously. This is in contrast to CPUs, which have fewer cores optimized for sequential processing.
- Key Features: Massive Parallelism, the ability to perform many operations simultaneously.
- High Throughput: Handles a high number of tasks concurrently, improving performance on workloads involving numerous small, independent calculations.

# What is a GPU, and why is it essential for deep learning?

## GPU vs. CPU: Key Differences

- Parallel vs. Serial Processing: GPUs excel at parallel tasks, while CPUs are designed for sequential tasks.
- Architecture: A CPU typically has 4-16 cores, optimized for high-speed execution of individual tasks, while a GPU may have thousands of smaller cores designed to handle numerous, concurrent operations.
- Memory Bandwidth: GPUs have higher memory bandwidth compared to CPUs, allowing faster data access, which is crucial for **deep learning models that process large datasets**.

# What is a GPU, and why is it essential for deep learning?

## Why GPUs Are Essential for Deep Learning

Deep learning involves training complex neural networks on large datasets, requiring millions or even billions of calculations.

These operations are often repetitive, involving **matrix multiplications and additions**, which are computationally intensive but **highly parallelizable**.

# What is a GPU, and why is it essential for deep learning?

## Why GPUs Are Essential for Deep Learning

- Speed: GPUs accelerate training times by handling multiple computations simultaneously. A task that might take days on a CPU can often be completed in hours on a GPU.
- Scalability: The parallel architecture of GPUs allows them to scale easily with the complexity and size of neural networks. As deep learning models grow larger, with more layers and neurons, GPUs can handle this increased workload efficiently.

# What is a GPU, and why is it essential for deep learning?

## Why GPUs Are Essential for Deep Learning

- Specialized Libraries: Many deep learning frameworks (e.g., TensorFlow, PyTorch) leverage GPU-optimized libraries like CUDA (Compute Unified Device Architecture) and cuDNN (CUDA Deep Neural Network library), which are specifically designed to maximize GPU performance.
- Support for Large Data: Deep learning often involves processing vast amounts of data (e.g., images, videos, text), and GPUs are well-suited for this because of their high memory bandwidth and capacity to handle large-scale data processing.

# What is a GPU, and why is it essential for deep learning?

## Applications of GPUs in Deep Learning

- Computer Vision: GPUs enable rapid training of convolutional neural networks (CNNs) for tasks like image classification, object detection, and segmentation.
- Natural Language Processing (NLP): GPUs are used to train recurrent neural networks (RNNs) and transformers for tasks like language translation, and text generation.
- Generative Models: Training of generative models like GANs (Generative Adversarial Networks) and VAEs (Variational Autoencoders) relies heavily on the computational power of GPUs.
- ...

# What is a GPU, and why is it essential for deep learning?

**Conclusion** GPUs are essential for deep learning due to their ability to handle the massive parallel processing requirements of neural networks.

By significantly reducing training time, scaling with model complexity, and efficiently managing large datasets, GPUs have become the backbone of modern deep learning applications.

Their use has democratized access to powerful AI models, enabling rapid advancements in fields ranging from autonomous vehicles to medical imaging and beyond.

# Overview of major GPU providers: NVIDIA, AMD.

**NVIDIA:**

Market Leader: Dominates AI and deep learning sectors with cutting-edge technologies.

Key Technologies:

- CUDA: Parallel computing platform optimized for deep learning.

- cuDNN: Accelerated library for neural networks.

- TensorRT: Inference optimizer for AI deployment.

# Overview of major GPU providers: NVIDIA, AMD.

**AMD:**

Challenger: Known for high-performance, cost-effective GPUs.

Focus on open-source solutions, competitive performance, and cloud collaborations.

Key Technologies:

- ROCm: Open-source platform for high-performance computing.

- MIOpen: Library for deep learning acceleration on AMD GPUs.

# Introduction to GPU computing libraries (CUDA, cuDNN).

**CUDA (Compute Unified Device Architecture)**

Developed by NVIDIA, CUDA is a parallel computing platform and application programming interface (API) model.

- Parallel Processing: Enables the execution of thousands of threads simultaneously, significantly speeding up tasks like matrix operations.

- Ease of Use: Integrates with popular programming languages (C, C++, Python, Fortran), making it accessible for developers.

- Broad Ecosystem: Supports major deep learning frameworks (TensorFlow, PyTorch)

# Introduction to GPU computing libraries (CUDA, cuDNN).

**cuDNN (CUDA Deep Neural Network library)**

Developed by NVIDIA, cuDNN is a GPU-accelerated library specifically designed for deep learning.

Provides highly optimized primitives for deep neural network building blocks.

- Optimized Routines: Includes high-performance implementations of forward and backward convolution, activation functions, pooling, and normalization.

- Compatibility: Compatible with popular deep learning frameworks such as TensorFlow, PyTorch, and Caffe.

## Synergy of CUDA and cuDNN in AI Workflows

Real-World Applications:

Used in training state-of-the-art models in computer vision (e.g., CNNs for image classification), natural language processing (e.g., transformers), and generative models (e.g., GANs).

Supports rapid prototyping and deployment, allowing organizations to move from research to production with minimal code changes.

Conclusion: CUDA and cuDNN are foundational technologies in GPU-accelerated deep learning.

# Hands on

## CPU vs. GPU: A Practical Comparison

download the file `Lab_1.py` from the course repository and complete it.

# Introduction to Dataloaders

## or how to load and preprocess data

### What are Dataloaders?

Definition: Dataloaders are utilities used to efficiently load and preprocess data batches for machine learning models. Importance: They help streamline the process of preparing data, ensuring efficient and scalable training.

# Role in Deep Learning Pipelines

**Purpose:**

- Efficient Loading: Ensure data is loaded and made available in a way that does not bottleneck the training process.

- Preprocessing: Perform necessary transformations on data, such as normalization and augmentation.

**Pipeline Placement:**

- Data Collection: Raw data is gathered.

- Dataloader: Data is passed through a Dataloader for batching and preprocessing.

- Model Training: Batches are fed to the model for training.

# Optimisation for GPUs

**Challenges with Large Datasets:**

- Memory Limitations: GPUs have limited memory which can be quickly exhausted with large datasets.

- Data Transfer Bottlenecks: Moving data between CPU and GPU can become a bottleneck.

Solutions:

- Efficient Data Loading: Implement techniques to reduce the overhead of loading data.

-Asynchronous Loading: Use asynchronous processes to preload data while the GPU is training.

# Concepts: Batch Processing

**What is Batch Processing?**

Definition: Splitting the dataset into smaller batches that are processed sequentially.

Advantages:

- Efficient Computation: Allows parallel processing on GPUs, making better use of computational resources.

- Hardware Utilization: Reduces the overhead of processing large datasets in one go.

# Concepts: Shuffle

**What is Shuffling?**

Definition: Randomly reordering the data before feeding it into the model.

Why Shuffle?

- Avoiding Bias: Prevents the model from learning patterns based on the order of the data.

- Improving Generalization: Helps the model generalize better by providing varied training samples.

# Concepts: Data Augmentation

**What is Data Augmentation?**

Definition: Techniques used to artificially increase the size of a dataset by applying transformations (e.g., rotations, translations).

Benefits:

- Increasing Dataset Size: Helps in creating more training samples from the existing data.

- Improving Robustness: Makes the model more robust to variations in the input data.

# Best Practices

**Efficient Data Loading Tips:**

- Use Efficient Formats: Store data in formats that are quick to read (e.g., HDF5, TFRecord).

- Minimize Overhead: Reduce the amount of preprocessing done on the fly to avoid bottlenecks.

Common Pitfalls:

- Memory Overload: Avoid loading too much data into memory at once.

- Inefficient Transfers: Ensure data transfer between CPU and GPU is optimized to prevent slowdowns.

# Summary

**Recap of Key Points:**

- Role of Dataloaders: Key component in managing data for training models.

- Concepts: Understanding batch processing, shuffling, and augmentation.

- Optimisation: Techniques to improve data loading efficiency and GPU utilization.

# Hands on

**Data Loading and Preprocessing in PyTorch**

download the file `Lab_2.py` from the course repository and complete it.

# Overview of Keras, PyTorch, and PyTorch Lightning

## or how to choose the right framework for your DL project

# Introduction to Deep Learning Frameworks

**Purpose of Frameworks:**

- Simplify the development and deployment of deep learning models.

- Provide pre-built functions, layers, and tools for building neural networks.

**Popular Frameworks:**

- Keras: User-friendly, high-level API often used with TensorFlow.

- PyTorch: Flexible, dynamic framework popular in research and production.

- PyTorch Lightning: A lightweight wrapper around PyTorch for cleaner code and easier experimentation.

# Keras - Key Features and Use Cases

**Keras Overview:**

- High-level API: Designed to be user-friendly, modular, and easy to extend.

- Integration: Often used as an API within TensorFlow but can work independently.

**Key Features:**

- Simple and Intuitive Syntax: Great for beginners and rapid prototyping.

- Extensive Pre-trained Models: Easy access to models like VGG, ResNet, and more via keras.applications.

- High-Level Abstractions: Simplifies model building, training, and evaluation.

# Keras - Key Features and Use Cases

**Use Cases:**

- Ideal for rapid prototyping and educational purposes.

- Widely used in small to medium-sized projects where ease of use is prioritized.

- Suitable for model deployment through TensorFlow Serving.

# PyTorch - Key Features and Use Cases

**PyTorch Overview:**

- Dynamic Computation Graphs: Changes during runtime, providing more flexibility and debugging ease.

- Pythonic: Feels natural for Python developers, offering direct integration with Python data structures.

**Key Features:**

- Autograd: Automatic differentiation for gradient computation.

- Rich Ecosystem: Strong community support with libraries like torchvision for images and torchaudio for audio processing.

- Extensive Control: Allows granular control over training loops, model layers, and optimizations.

# PyTorch - Key Features and Use Cases

**Use Cases:**

- Preferred for research and development due to its flexibility.

- Commonly used in computer vision, NLP, and reinforcement learning.

- Industry adoption: Increasingly used in production environments due to its performance and community support.

# PyTorch Lightning - Key Features and Use Cases

**PyTorch Lightning Overview:**

- Wrapper for PyTorch: Simplifies training loops while retaining PyTorch's flexibility.

- Structured Code: Enforces a clear separation between model definition, training, and validation.

**Key Features:**

- Reduce Boilerplate: Automatically handles training, validation, and testing logic.

- Scalable Training: Easy integration with multi-GPU, TPU, and distributed training setups.

- Logging and Callbacks: Built-in support for logging (TensorBoard, WandB) and customizable callbacks.

# PyTorch Lightning - Key Features and Use Cases

**Use Cases:**

- Ideal for academic research with complex models needing rapid iteration.

- Used in production to reduce code complexity and improve model reproducibility.

- Facilitates experimentation with hyperparameter tuning and multi-GPU setups.

# Main Differences Between Keras, PyTorch, and PyTorch Lightning

**Ease of Use:**

- Keras: Easiest for beginners; minimal code needed for training.

- PyTorch: Flexible, but requires more manual handling of training loops.

- PyTorch Lightning: Eases PyTorch development with structured code.

**Flexibility:**

- Keras: High-level, but can be limiting for complex custom models.

- PyTorch: Full control over all aspects of the training process.

- PyTorch Lightning: Balances flexibility with ease of use, keeping control within the model definition.

# Keras Project Structure

**Typical Keras Project Layout:**

- Data Preparation: Load and preprocess data using tf.keras.preprocessing.

- Model Definition: Define the model architecture using the Sequential or Functional API.

- Compilation: Compile the model with optimizer, loss, and metrics.

- Training: Use `model.fit()` to train the model with callbacks for monitoring.

- Evaluation and Prediction: Evaluate model performance using `model.evaluate()` and make predictions.

# Keras Project Structure

**Example Structure:**

```
|-- data.py           # Data loading and preprocessing
|-- model.py           # Model definition
|-- train.py           # Training script
|-- evaluate.py        # Evaluation script
|-- requirements.txt # Dependencies
```

# PyTorch Project Structure

**Typical PyTorch Project Layout:**

- Data Loading: Use `torch.utils.data` for loading and preprocessing data.

- Model Definition: Define the model class inheriting from `nn.Module`.

- Training Loop: Write custom training, validation, and testing loops.

- Loss and Optimizer: Define loss functions and optimizers (e.g., `torch.optim.SGD`).

- Checkpoints: Save and load model checkpoints during training.

# PyTorch Project Structure

**Example Structure:**

```
|-- data_loader.py # Data loading and transformations
|-- model.py # Model definition (nn.Module)
|-- train.py # Training and validation loops
|-- test.py # Testing and inference
|-- utils.py # Utility functions (metrics, logging)
|-- config.yaml # Configuration file for hyperparameters
```

# PyTorch Lightning Project Structure

**Typical PyTorch Lightning Project Layout:**

- Data Module: Use pl.LightningDataModule to handle data loading and preprocessing.

- Lightning Module: Define model logic, training steps, validation steps, and testing steps.

- Trainer: Use Trainer to handle training, validation, testing, and scaling.

- Callbacks: Use callbacks for early stopping, logging, and checkpointing.

# PyTorch Lightning Project Structure

**Example Structure:**

```
|-- data_module.py # LightningDataModule for data handling
|-- model.py # LightningModule for model definition
|-- train.py # Training and testing script using Trainer
|-- callbacks.py # Custom callbacks (e.g., checkpointing)
|-- config.yaml # Configuration file for hyperparameters
|-- requirements.txt # Dependencies
```

# Choosing the Right Framework

**When to Use Keras:**

- Best for beginners and quick prototyping.

- Suitable for projects that do not require complex customization.

**When to Use PyTorch:**

- Ideal for research and projects requiring full control over model design and training.

- Preferred in academia due to its dynamic nature and debugging ease.

**When to Use PyTorch Lightning:**

- Great for scaling PyTorch models with less code complexity.

- Suitable for researchers and production environments needing rapid iteration and clean code.

# Conclusion and Best Practices

**Framework Choice:**

- No single best framework; it depends on project requirements and team expertise.
- Leverage Keras for simplicity, PyTorch for flexibility, and PyTorch Lightning for structured PyTorch development.

**Best Practices:**

- Modular Code: Keep data, model, and training logic separate.
- Version Control: Use version control (`Git`) for tracking experiments.
- Scalability: Consider scalability needs early in development.

**Final Thoughts:**

- Mastering multiple frameworks enhances versatility and allows for optimal tool selection based on specific project needs.

# Hands on

**Data Loading and Preprocessing in PyTorch**

download the file `Lab_3.py` from the course repository and complete it.