IT UNIVERSITY OF COPENHAGEN

ML for Systems, Systems for ML

Ehsan Yousefzadeh-Asl-Miandoab

ehyo@itu.dk

IoT course - 2026

Content

- Programming vs. Machine Learning
- Machine Learning Methods
- Resource Constraints in IoT systems
- Model Optimization Techniques for IoT systems
- A real example of a ML-based solution for a system
  - *Data collection is the most important and most expensive phase!*

Some slides are for later reading (Reading Assignments).

Search and read more about unfamiliar keywords.

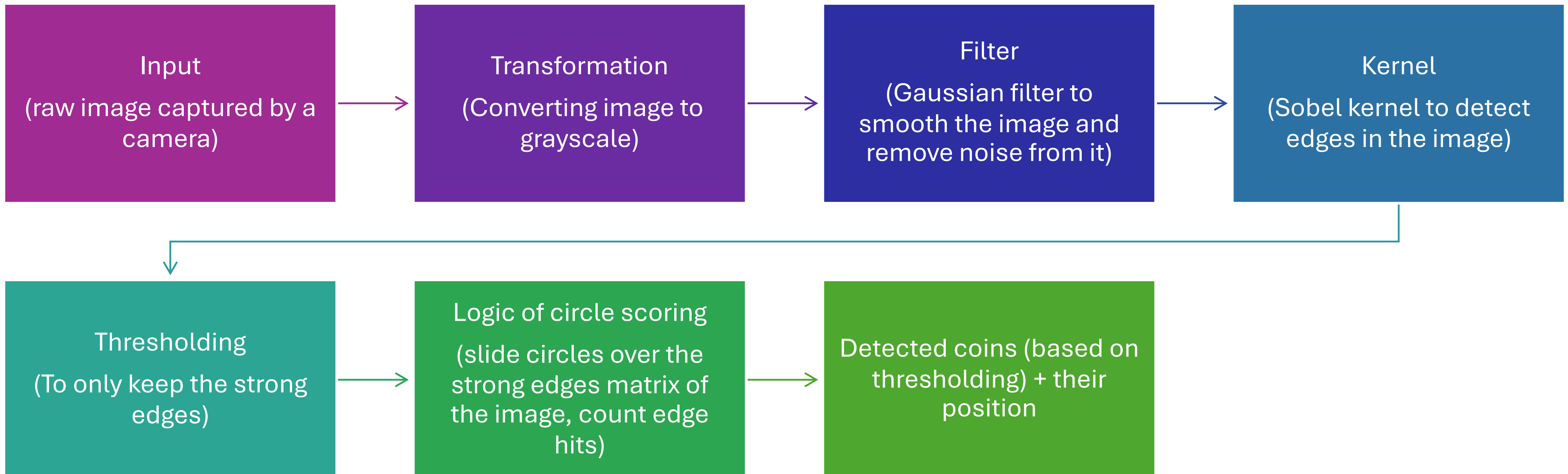
Slides with the "Hands-On" tag indicate that the teacher will dive into the code and execute it live during the class.

What is Programming?

- A human (programmer) encodes knowledge as explicit rules
- Input Data + Rules → Output

How to detect coins?

If I am the programmer!



RGB vs. Grayscale images and how we change an RGB to a grayscale

- An **RGB image** stores each pixel with three values: **Red, Green, and Blue**. By combining these three channels, the image can represent many colors. A **grayscale image** stores only one intensity value per pixel, which shows how dark or bright that pixel is, from black to white. This makes grayscale images simpler and cheaper to store and process.
- To convert an RGB image to grayscale, we usually do **not** just average the three channels equally. Instead, we use a **weighted combination** because human vision is more sensitive to green than to red, and more sensitive to red than to blue. A common formula is:

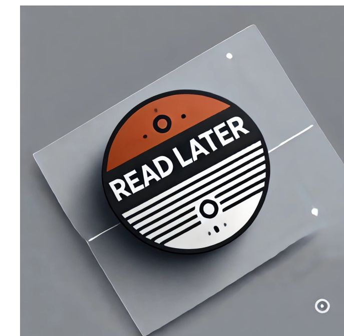
$$\textbf{Gray} = \textbf{0.299 R + 0.587 G + 0.114 B}$$

- This keeps the brightness closer to what people actually perceive in the original color image. In practice, grayscale conversion is useful in many vision tasks such as preprocessing, edge detection, segmentation, and reducing computation.
- **Read more:** OpenCV color conversion documentation: https://docs.opencv.org/3.4/de/d25/imgproc_color_conversions.html

Noise in an image & Gaussian Filter

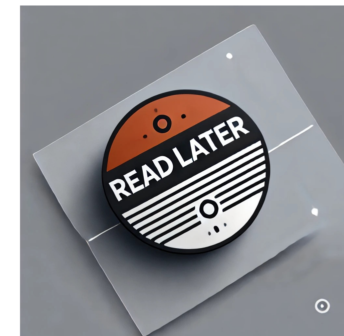
- **Image noise** is unwanted random variation in pixel values that makes an image look grainy, speckled, or less clear. Noise can come from the camera sensor, poor lighting, high ISO settings, transmission errors, or image compression. In image processing, noise matters because it can hide important details and reduce the accuracy of later steps such as edge detection, segmentation, or object recognition.
- A **Gaussian filter** is a smoothing filter used to reduce this noise. It works by replacing each pixel with a weighted average of nearby pixels, where closer pixels get higher weight according to a Gaussian distribution. This reduces high-frequency variation, so random noise becomes weaker. The tradeoff is that strong smoothing can also blur edges and fine details. In practice, Gaussian filtering is often used as a first preprocessing step before more advanced vision tasks. OpenCV describes Gaussian blur as convolution with a Gaussian kernel and notes that image blurring is useful for removing noise.
- **Read more:** OpenCV tutorial on smoothing images: https://docs.opencv.org/4.x/d4/d13/tutorial_py_filtering.html

Edge Detection



- Edge detection is the process of finding points in an image where the intensity changes sharply. These changes often correspond to object boundaries, shape outlines, texture changes, or important structure in the scene. Instead of keeping all pixel information, edge detection focuses on the most meaningful transitions, which makes later tasks such as segmentation, contour extraction, and object recognition easier. OpenCV describes Canny as a popular multi-stage edge detector that first reduces noise, then computes image gradients, suppresses weak non-edge responses, and finally keeps edges using thresholding.
- A very common method is **Canny edge detection**. It usually starts with a **Gaussian filter** to reduce noise, because noise can create false edges. Then it measures intensity change in the horizontal and vertical directions, computes edge strength and direction, thins the edges, and applies thresholding to keep the most meaningful boundaries. This is why edge detection is often taught together with smoothing: good edge results depend strongly on noise reduction first.
- **Read more:** OpenCV tutorial on Canny edge detection: https://docs.opencv.org/4.x/da/d22/tutorial_py_canny.html

Sobel Kernel



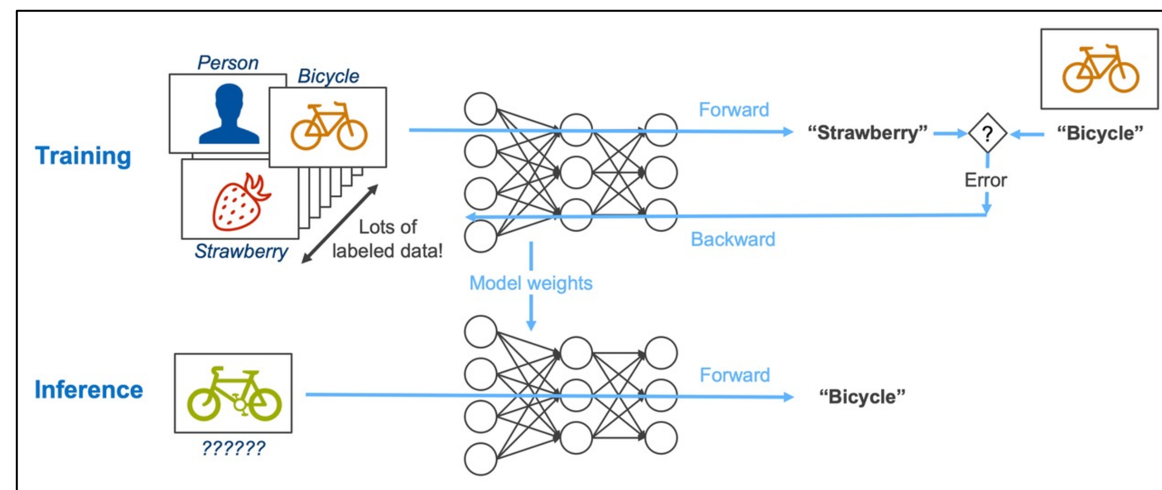
- The **Sobel kernel** is a small matrix used to estimate how quickly image intensity changes from one pixel to the next. It is mainly used to detect **edges** by computing image gradients in the horizontal and vertical directions. In practice, one kernel is applied to detect change in the **x-direction**, and another to detect change in the **y-direction**. Large responses usually indicate an edge or boundary in the image. OpenCV describes Sobel as a derivative-based operator for calculating image gradients, and notes that it combines differentiation with smoothing, which makes it more resistant to noise than a plain derivative.
- A common pair of Sobel kernels is:

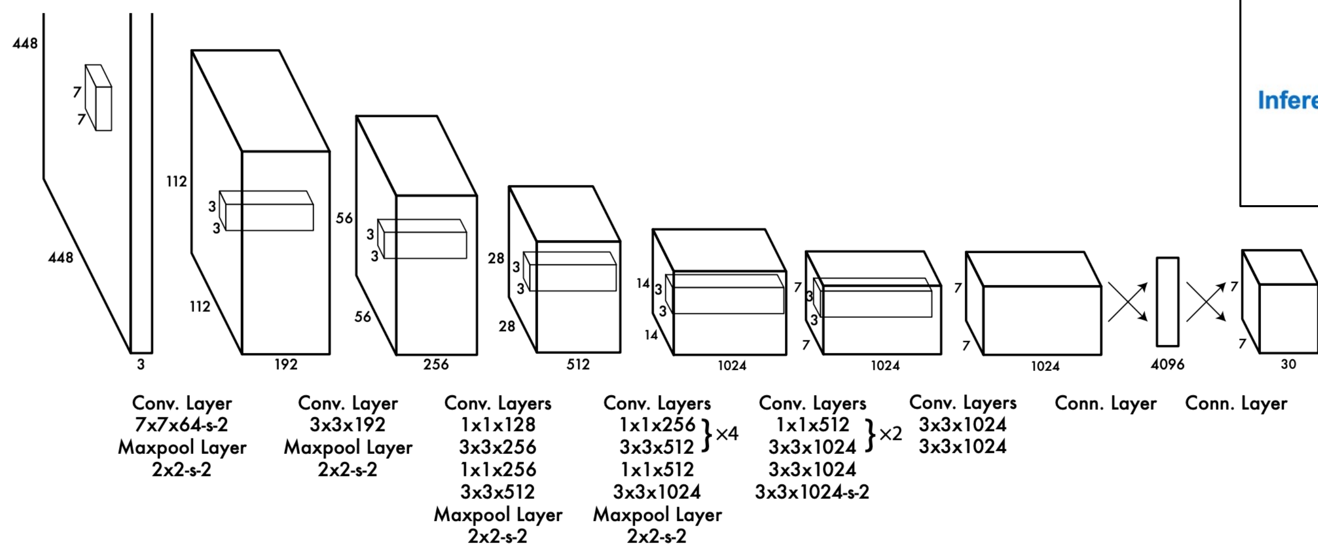
$$G_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \; G_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

- These kernels highlight vertical and horizontal intensity changes, and their outputs can be combined to estimate edge strength. The Sobel operator is widely used because it is simple, fast, and effective for introducing gradient-based edge detection, though stronger noise or fine-detail tasks may need more advanced methods.
- **Read more:** OpenCV Sobel derivatives tutorial. https://docs.opencv.org/4.x/d2/d2c/tutorial_sobel_derivatives.html

What is Machine Learning?

- A system learns rules from data
- Input + Output → Rules



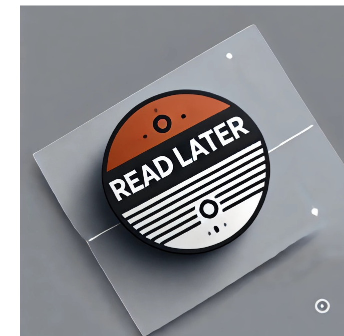


https://arxiv.org/pdf/1506.02640

https://www.linkedin.com/pulse/difference-between-deep-learning-training-inference-mark-robins-mdq8c

https://universe.roboflow.com/guilherme-dos-santos-novak-kjqca/coins-detection-cxmsx

YOLO model



- The **YOLO (You Only Look Once)** model is a family of deep learning models for **real-time object detection**. Its goal is to look at an image only once and directly predict **what objects are present** and **where they are located** using bounding boxes. This is different from older approaches that first generated many candidate regions and then classified them. The original YOLO paper describes detection as a single regression problem from image pixels to bounding boxes and class probabilities, which made it fast enough for real-time use.
- Modern YOLO systems are widely used because they balance **speed and accuracy** well, which is why they appear in applications such as traffic monitoring, robotics, surveillance, industrial inspection, and autonomous systems. In practice, a YOLO detector outputs bounding boxes, class labels, and confidence scores for the detected objects. Current Ultralytics YOLO documentation also shows that the YOLO family is used not only for detection, but also for related tasks such as segmentation, pose estimation, tracking, and classification.
- **Read more:** Ultralytics YOLO documentation. https://docs.ultralytics.com/

MLPs

- An **MLP (Multilayer Perceptron)** is a neural network made of several layers of neurons, usually arranged as an **input layer**, one or more **hidden layers**, and an **output layer**. Information moves forward layer by layer, so it is a type of **feedforward network**. In many practical models, the hidden layers in an MLP are built from **fully connected (FC)** layers plus nonlinear activation functions such as ReLU. PyTorch describes a feedforward network as passing the input through several layers one after another, and its Linear layer as applying an affine transformation to the input.
- MLPs are commonly used when the input is represented as a feature vector and there is no strong spatial or sequential structure to exploit. The Deep Learning book describes deep feedforward networks as multilayer perceptrons, and Goodfellow's teaching material also presents a "fully connected baseline" as a 2–3 hidden-layer feedforward network, also called an MLP.
- Read more:
  - https://docs.pytorch.org/tutorials/beginner/blitz/neural_networks_tutorial.html
  - https://www.deeplearningbook.org/

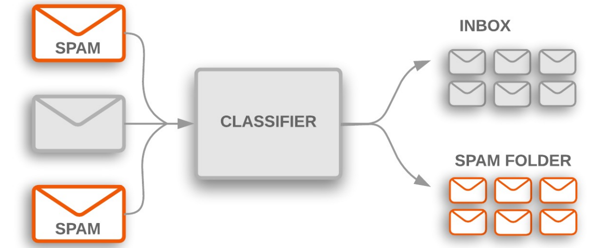
CNNs

- A **Convolutional Neural Network (CNN)** is a neural network designed especially for data with grid-like structure, such as images. Instead of connecting every pixel to every neuron, a CNN uses **small filters (kernels)** that slide over the image and learn local patterns such as edges, corners, and textures. Deeper layers then combine these simple patterns into more complex features, such as shapes or object parts. The Deep Learning book describes convolutional networks as a specialized kind of neural network for processing data with a known grid-like topology, such as image data.
- A typical CNN contains **convolution layers**, **activation functions** such as ReLU, and often **pooling layers** that reduce spatial size while keeping important information. This structure helps CNNs use fewer parameters than fully connected networks on images, while also taking advantage of spatial locality. In PyTorch's neural network examples, convolution is described as combining each image element with its nearby neighbors through a kernel to extract useful features like edges or blur patterns (Read More: https://docs.pytorch.org/tutorials/recipes/recipes/defining_a_neural_network.html).
- CNNs are widely used in **image classification, object detection, segmentation, medical imaging, autonomous systems, and video analysis** because they are effective at learning visual features automatically from data. They became one of the most important model families in computer vision for this reason.
- **Read more:** Deep Learning Book, Chapter 9 on Convolutional Networks. https://www.deeplearningbook.org/contents/convnets.html

Transformers

- A **Transformer** is a neural network architecture designed to process sequences, such as text, by using **attention** to decide which parts of the input are most relevant to each other. Unlike older recurrent models, Transformers do not process tokens one by one in strict order. Instead, they can look at many positions in parallel, which makes training more efficient. The original *Attention Is All You Need* paper introduced the Transformer as a model based only on attention mechanisms, without recurrence or convolution (Read more: https://arxiv.org/abs/1706.03762).
- The key idea is **self-attention**. For each word or token, the model compares it with other tokens in the sequence and learns how much attention to give to each one. This helps the model capture relationships such as grammar, meaning, and long-range dependencies. A standard Transformer also includes **multi-head attention**, **feedforward layers**, and **positional encoding**, which lets the model keep track of token order (Read more: https://docs.pytorch.org/docs/stable/generated/torch.nn.Transformer.html).
- Transformers are now the foundation of many modern AI systems, especially in **large language models**, but they are also used in vision, speech, and multimodal learning. They became important because they scale well and work very effectively on large datasets.

Types of Machine Learning



- Supervised
  - Classification
  - Regression
    - Predict a student's **final exam score** from homework, quizzes, and attendance.

- Unsupervised
  - the computer tries to find patterns on its own.
    - You collect data about students' study habits, attendance, and assignment patterns, but you do **not** label them. Clustering may reveal groups such as:
    - highly consistent students
    - last-minute learners
    - students needing support

- Reinforcement learning
  - an agent learns by **trial and error** and gets **rewards or penalties** based on its actions.
    - self-driving decisions, robot navigation, game-playing AI

Clustering as an unsupervised method

- **Clustering** is an **unsupervised learning** method, which means it works on **unlabeled data**. Instead of learning from correct answers, the algorithm tries to discover natural groups in the data by finding samples that are similar to each other and separating them from samples that are different. This makes clustering useful when we do not already know the classes in advance. Scikit-learn places clustering under unsupervised learning, and IBM describes it as grouping unlabeled data based on similarities or differences (https://scikit-learn.org/stable/unsupervised_learning.html).
- The goal of clustering is to organize data into **clusters**, where points inside the same cluster are more alike than points in different clusters. Common examples include **customer segmentation**, **grouping similar images**, **document organization**, and sometimes **anomaly detection** when some points do not fit well into any cluster. Different clustering methods make different assumptions: for example, **K-means** looks for compact groups, while methods such as **DBSCAN** can find clusters with irregular shapes (https://scikit-learn.org/stable/modules/clustering.html).
- You should understand that clustering does not give “correct labels” by itself. Instead, it helps us explore structure in data, and then we interpret what each cluster may mean. That is why clustering is often used for **data exploration** as a first step before building more advanced machine learning systems.

Where do we train and do inference?

- Training is done on big GPU/TPU servers in data centers or the cloud
- Inference can be done in the cloud, local server, or on the edge device depending on:
  - Latency
  - Cost
  - Privacy
  - Bandwidth
  - Device compatibility

Edge inference is possible, but not always possible.
Because many edge devices have limited compute, memory, and power.

CPU/ GPU/ TPU

- A **CPU (Central Processing Unit)** is the general-purpose processor of a computer. It handles many different kinds of tasks, runs the operating system, and is built to be flexible and good at sequential control-heavy work. Intel describes the CPU as essential to modern computing systems because it executes the commands and processes needed for the computer and operating system.
- A **GPU (Graphics Processing Unit)** is designed for highly parallel work, where many operations can be done at the same time. That makes GPUs especially useful for graphics, scientific computing, and deep learning. NVIDIA describes the GPU as the technology that drives AI, high-performance computing, graphics, and robotics, and its CUDA documentation explains that different GPU architectures provide hardware features for accelerated computation.
- A **TPU (Tensor Processing Unit)** is a specialized accelerator made mainly for machine learning. Google describes TPUs as custom-developed ASICs optimized for training and inference of AI models, especially for fast matrix operations that appear often in neural networks. Compared with CPUs and GPUs, TPUs are more specialized and are mainly used in large-scale AI workloads rather than general-purpose computing.

https://www.intel.com/content/www/us/en/products/docs/processors/cpu-vs-gpu.html
https://www.nvidia.com/en-us/
https://cloud.google.com/tpu

Assignment

- Choose a dataset from here
  - https://universe.roboflow.com/search?q=like-tag:aerial
  - Choose a YOLO model
  - Train it on your laptop
    - If you do not have GPU, use university's HPC
      - http://hpc.itu.dk/
  - Gather a set of images and show how your model detects
  - Read and understand YOLO metrics: https://docs.ultralytics.com/guides/yolo-performance-metrics/#introduction
- Think about a logic that you can make rules for detecting those objects!
- Stay curious and keep those questions coming!

Training LLMs is very expensive!

- Llama 3.1 / 405B parameters
  - 16,384 NVIDIA H100 GPUs
  - 54 days for pre-training

- PaLM / 540B
  - 6,144 TPU v4 chips
  - ~ 64 days

Training usually needs huge clusters.

https://ar5iv.labs.arxiv.org/html/2407.21783

https://arxiv.org/pdf/2204.02311

What about inference costs?

- LLMs' inference is expensive

- For **Llama 3.1**,
  - **8B model**: about **16 GB VRAM – workstation/ server GPU**
  - **70B model**: about **140 GB VRAM – one or few high memory GPUs**
  - **405B model**: about **810 GB VRAM – multi-GPU server or cloud cluster**

- The main resource needs include:
  - GPU memory (VRAM) to hold model parameters
  - Extra memory for KV cache during generation
  - Enough compute throughout to produce tokens fast enough

LLM inference is hard to push on tiny edge devices!

How LLMs generate tokens?

- An LLM first **breaks your input into tokens**, which are small text units such as whole words, parts of words, punctuation, or spaces. OpenAI describes tokens as the building blocks models process, and notes that common words may be one token while longer words may be split into pieces.
- Then the model processes all input tokens with a **Transformer** and computes a **probability distribution for the next token**. In decoder-style language models, generation is **autoregressive**, meaning the model predicts one next token at a time, appends it to the sequence, and repeats the process. This is the core idea behind Transformer decoding.
- The next token is chosen using a **decoding strategy**. The simplest is **greedy decoding**, which picks the highest-probability token. Other methods, such as sampling, choose among likely tokens to make output less repetitive or more creative. Hugging Face's generation docs describe greedy search as selecting the most likely next token by default.
- This loop continues until the model produces an **end-of-sequence token** or reaches a length limit. So, in simple terms: **tokenize input → predict next token → add it → repeat**.

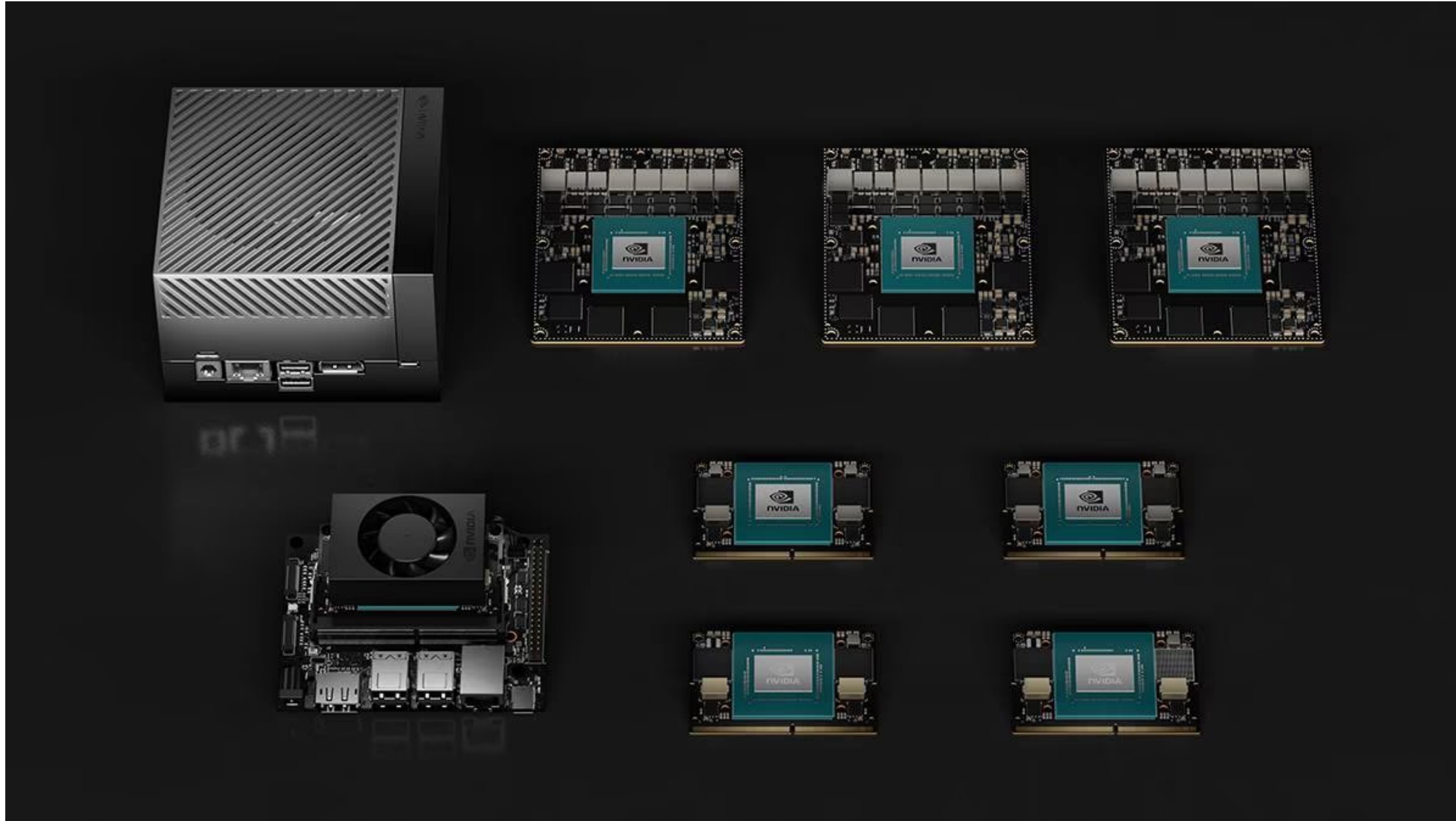
https://help.openai.com/en/articles/4936856-what-are-tokens-and-how-to-count-them
https://huggingface.co/docs/transformers/v4.36.1/llm_tutorial
https://huggingface.co/docs/transformers/v4.47.1/generation_strategies

KV cache in an LLM.

- The **KV cache** is a memory structure used during **LLM inference** to store the **keys (K)** and **values (V)** produced by the attention layers for tokens that have already been processed. Instead of recomputing these for the whole previous context every time a new token is generated, the model reuses the stored K and V and only computes the new token's attention information. Hugging Face describes KV cache exactly as storing key-value pairs from previously processed tokens so they can be reused for later tokens.
- This matters because LLM generation is **autoregressive**: the model produces one token, then the next, then the next. Without KV caching, each new step would repeatedly recompute attention over all earlier tokens, which becomes increasingly expensive as the sequence grows. With a KV cache, the model keeps the old attention state and extends it token by token, which makes decoding much faster.
- The main tradeoff is **speed versus memory**. KV caching reduces repeated computation and improves response speed, but it consumes memory that grows with sequence length, batch size, and model size. NVIDIA notes that, in practice, model weights and the KV cache are the two main contributors to GPU memory use during LLM inference.
- Imagine it like this: when reading a sentence word by word, the model keeps a compact memory of what it has already processed, so it does not need to "re-read" the whole sentence from scratch every time it generates the next word. Also, this cache is used for **inference**, not normally for training.

https://huggingface.co/docs/transformers/cache_explanation
https://developer.nvidia.com/blog/mastering-llm-techniques-inference-optimization

What about inference costs for YOLO? (p. 2)

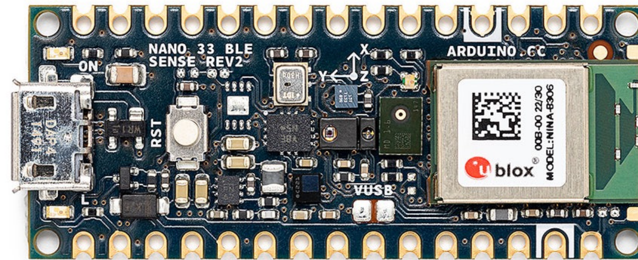


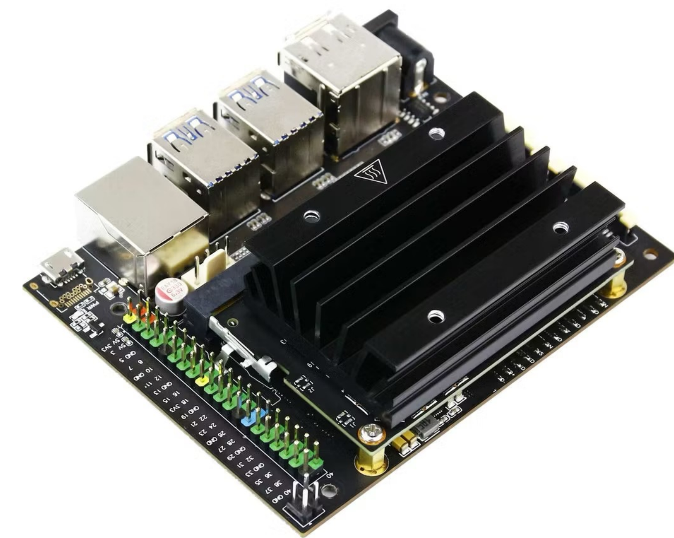
https://docs.ultralytics.com/guides/nvidia-jetson/

	Jetson AGX Thor(T5000)	**Jetson AGX Orin 64GB**	**Jetson Orin NX 16GB**	**Jetson Orin Nano Super**	**Jetson AGX Xavier**	**Jetson Xavier NX**	**Jetson Nano**
AI Performance	2070 TFLOPS	275 TOPS	100 TOPS	67 TOPS	32 TOPS	21 TOPS	472 GFLOPS
GPU	2560-core NVIDIA Blackwell architecture GPU with 96 Tensor Cores	2048-core NVIDIA Ampere architecture GPU with 64 Tensor Cores	1024-core NVIDIA Ampere architecture GPU with 32 Tensor Cores	1024-core NVIDIA Ampere architecture GPU with 32 Tensor Cores	512-core NVIDIA Volta architecture GPU with 64 Tensor Cores	384-core NVIDIA Volta™ architecture GPU with 48 Tensor Cores	128-core NVIDIA Maxwell™ architectur GPU
GPU Max Frequency	1.57 GHz	1.3 GHz	918 MHz	1020 MHz	1377 MHz	1100 MHz	921MHz
CPU	14-core Arm® Neoverse®-V3AE 64-bit CPU 1MB L2 + 16MB L3	12-core NVIDIA Arm® Cortex A78AE v8.2 64-bit CPU 3MB L2 + 6MB L3	8-core NVIDIA Arm® Cortex A78AE v8.2 64-bit CPU 2MB L2 + 4MB L3	6-core Arm® Cortex®-A78AE v8.2 64-bit CPU 1.5MB L2 + 4MB L3	8-core NVIDIA Carmel Arm®v8.2 64-bit CPU 8MB L2 + 4MB L3	6-core NVIDIA Carmel Arm®v8.2 64-bit CPU 6MB L2 + 4MB L3	Quad-Core Arm® Cortex®-A57 MPCore processor
CPU Max Frequency	2.6 GHz	2.2 GHz	2.0 GHz	1.7 GHz	2.2 GHz	1.9 GHz	1.43GHz
Memory	128GB 256-bit LPDDR5X 273GB/s	64GB 256-bit LPDDR5 204.8GB/s	16GB 128-bit LPDDR5 102.4GB/s	8GB 128-bit LPDDR5 102 GB/s	32GB 256-bit LPDDR4x 136.5GB/s	8GB 128-bit LPDDR4x 59.7GB/s	4GB 64-bi LPDDR4 25.6GB/s

https://docs.ultralytics.com/guides/nvidia-jetson/

Tiny devices limitations





	Arduino Nano 33 BLE Sense Rev2 (Tiny)	NVIDIA Jetson Nano (edge)
Processor	nRF52840, **64 MHz**	Quad-core ARM Cortex-A57, up to **1.43 GHz**
Main Memory	**256KB** SRAM	**4GB** LPDDR4
Storage	**1MB** flash	**16GB** eMMC
GPU	None	**128 core** GPU
Compute style	Tiny sensor/audio models	Full Linux edge AI, vision, larger DL models
Typical workloads	gesture recognition, anomaly detection	YOLO, video analytics, robotics, larger inference pipelines
Power	10X of mW	5-10 W

Typical Workloads on Tiny devices

- Keyword spotting / wake words
- Gesture / activity recognition
  - accelerometer/gyroscope data
- Anomaly detection
  - detecting unusual vibration, sound, temperature, or machine behavior.
- Simple image classification
  - classifying a small grayscale camera image into a few classes.
- Sound / audio classification
  - detecting claps, alarms, machinery states, or simple acoustic classes.

Optimizations enabling ML on tiny devices

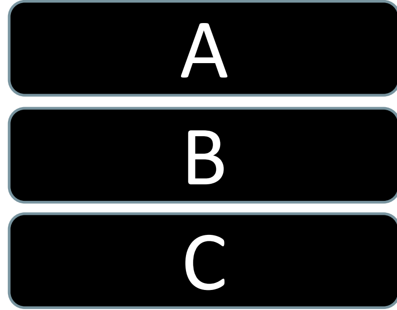
- **Quantization**
  - Use lower-precision numbers such as **int8 instead of float32** to reduce model size and enable faster fixed-point inference on embedded hardware.
- **Pruning / sparsity**
  - Remove less important weights so the model has fewer effective parameters and can need less storage and computation.
- **Knowledge distillation**
  - Train a smaller "student" model to imitate a larger model, keeping much of the accuracy with far less compute and memory.
- **Efficient model architectures**
  - Use models designed for tiny hardware rather than full-size networks; in practice this often means shallower/narrower networks or TinyML-specific architectures.
- **Input reduction and feature extraction**
  - Shrink the input or convert raw signals into compact features first, such as lower resolution/grayscale images, so the model processes less data.

End-to-End real-world ML-based solution example

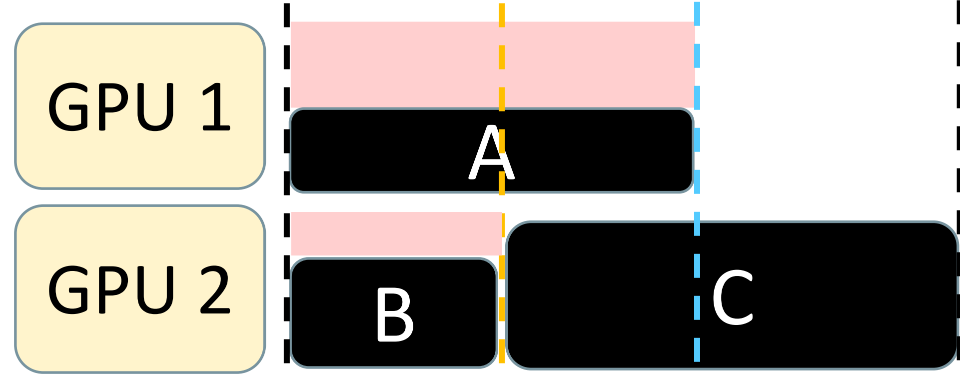
- Predicting GPU memory need of deep learning training tasks

- Industry reports show that GPUs are utilized for only ~50%.
  - Energy inefficient
  - Investment loss

Solution: Collocation of Model Training

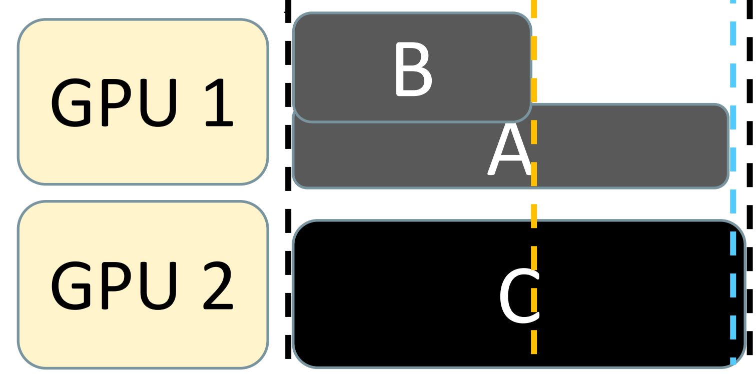


Conventional



- × **C experiences waiting time**
- × **Waste of energy and resources**

Collocation aware



- × **Slowdown due to resource interference**
- ✓ **Higher GPU utilization**
- ✓ **Higher throughput**

Gefion AI Supercomputer

1528 NVIDIA H100 Tensor Core GPUs

Reports from industry reveal 50% utilization of GPUs.

End-to-End real-world ML-based solution example

- Predicting GPU memory need of deep learning training tasks

Dataset Curations

Analyzing the dataset

Training the model

Using the model

Estimating GPU Memory is challenging!

- Optimizations
  - Applied by Default:
    - Activation reuse, dynamic memory management
  - May be enabled by the user:
    - Layer fusion, gradient checkpointing, mixed precision, etc.

These introduce levels of unpredictability to GPU memory estimation.

Existing Estimators

1. Analytical
   1. Horus formula (***TPDS ’21***)
   2. DNNMem, Microsoft’s Analytical work (***ESEC/FSE ’20***)
   3. LLMem (***IJCAI ’24***)

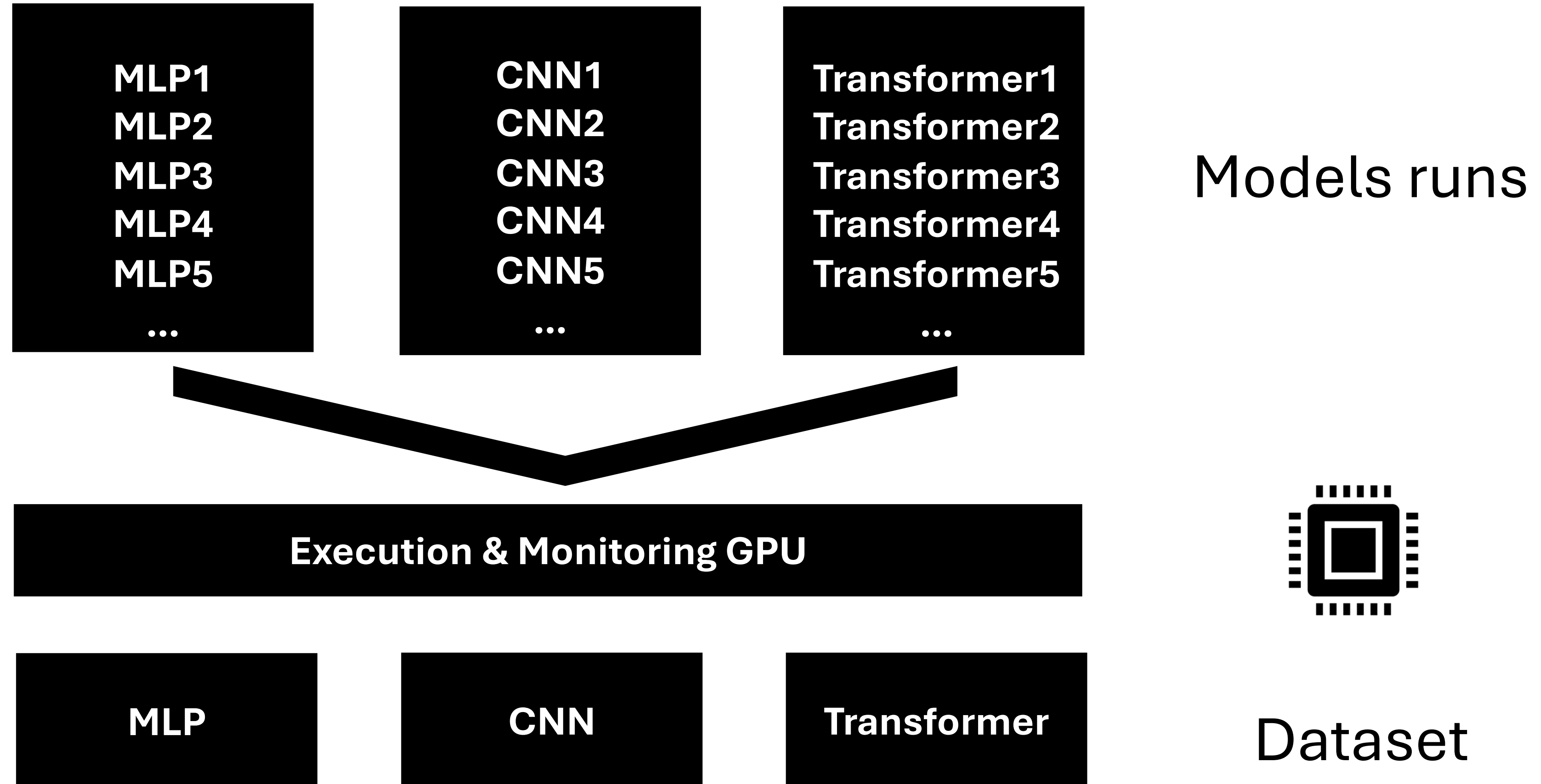
2. Libraries
   1. Fake Tensor
   2. DeepSpeed

3. ML-based approach
   - DNNPerf, graph neural networks (***ICSE-SEIP ‘23***)

https://pytorch.org/docs/stable/torch.compiler_fake_tensor.html
https://deepspeed.readthedocs.io/en/latest/memory.html

Challenges of Using ML for Estimation

Dataset



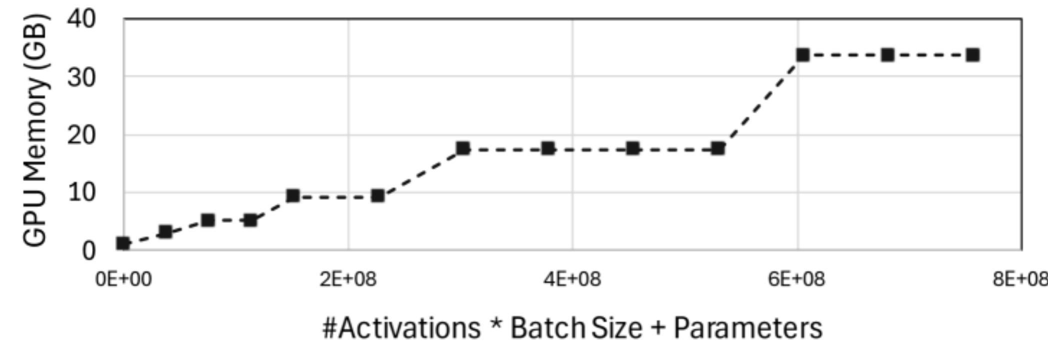
Models runs

Dataset

Considerations for Data Collection

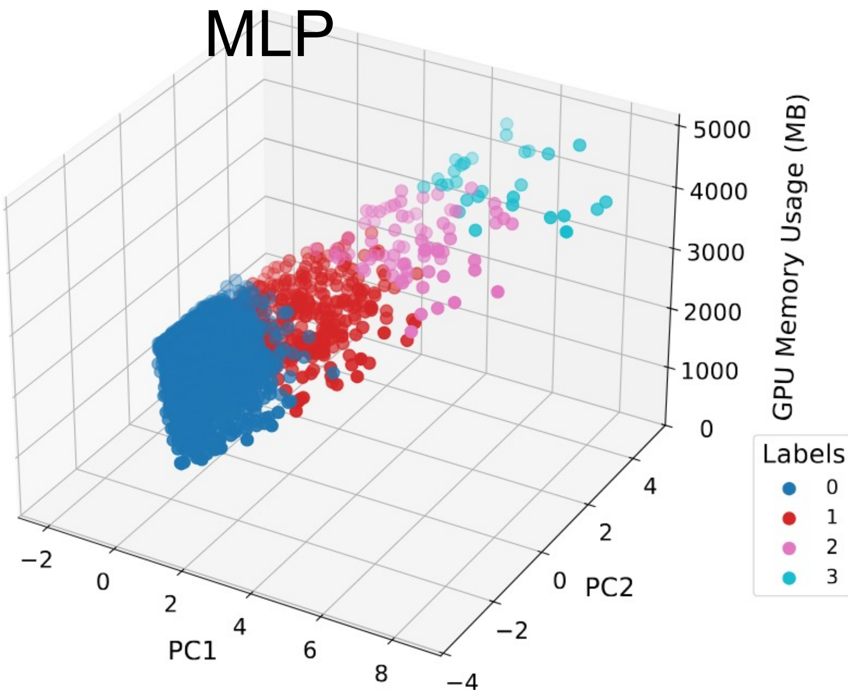
- **Architecture over model types.** To ensure long-term relevance, we focus on architecture families (e.g., CNNs, Transformers) rather than specific models, which quickly become outdated. This means that the data collection scripts must create synthetic versions of the architecture families.
- **Representative and realistic designs.** Synthetic models are generated with feasible architectures (e.g., avoiding excessively deep models) and realistic components such as dropout and batch normalization.
- **Diverse and balanced coverage.** We uniformly sample across input features and include varied topologies (e.g., pyramid, hourglass) to promote generalization.
- **Input/output variation.** Models include varying input and output dimensions to assess their impact on GPU memory usage.

Classification Formulation

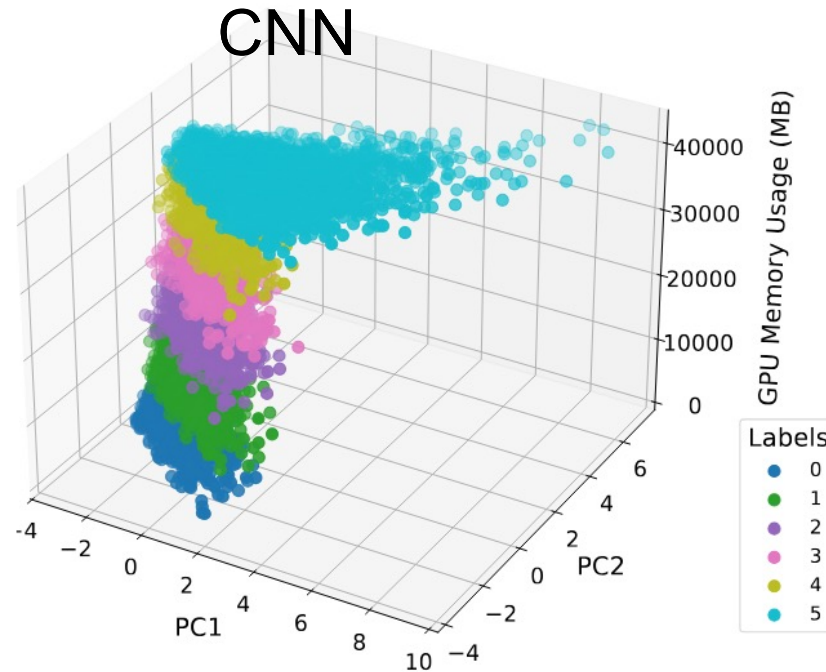


- Discretize data into same-GB classes
  - e.g., 0-1GB (1), 1GB-2GB (2), …
- Looked into the data through PCA and t-SNE
  - The classes are observable!

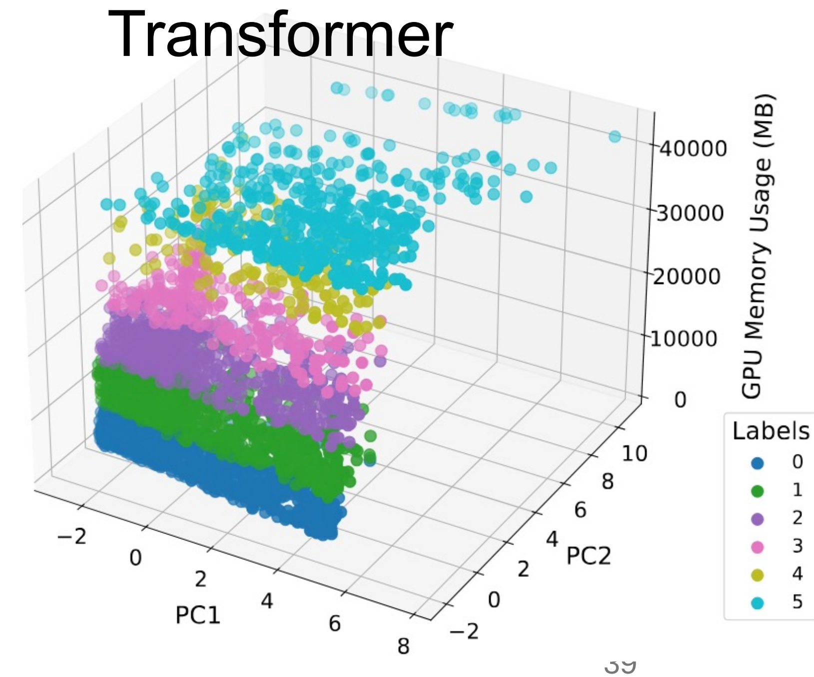
MLP



CNN

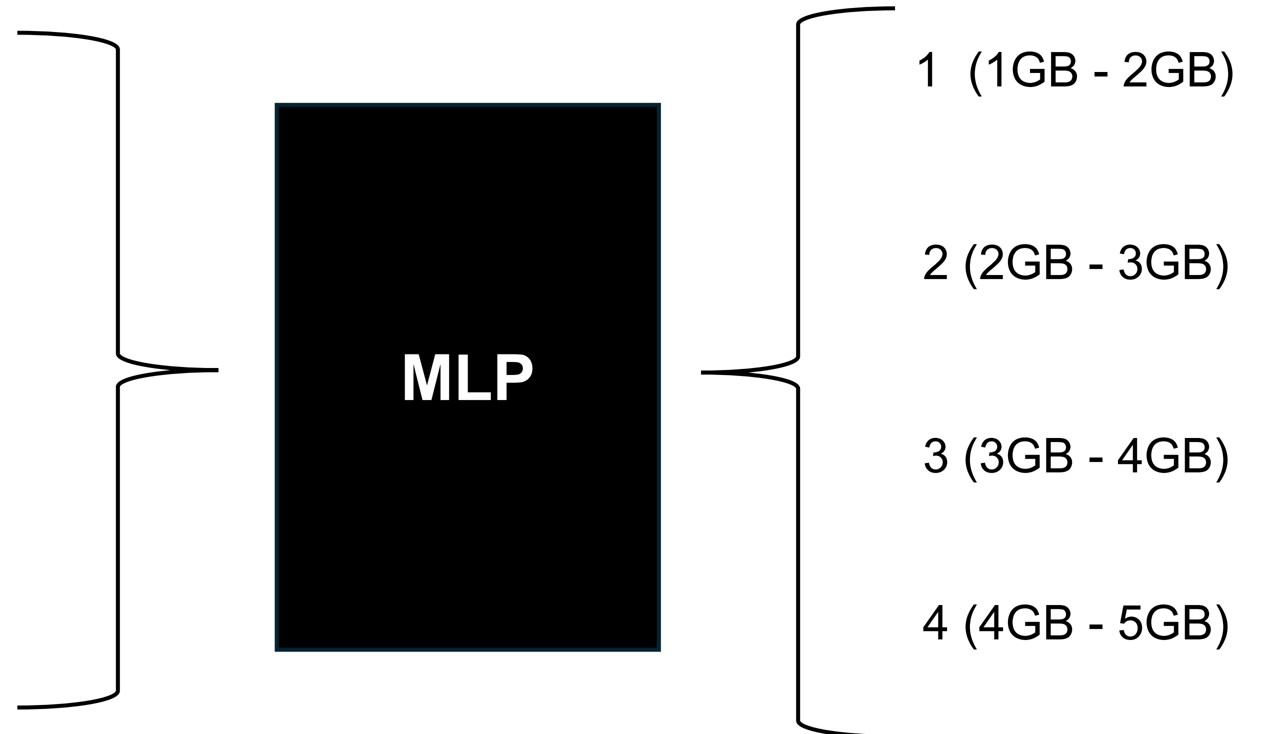


Transformer



Memory Estimator - MLP-based

- #linear layers
- #batch normalization layers
- #dropout layers
- batch size
- #parameters
- #activations
- activation encoding cos/sin

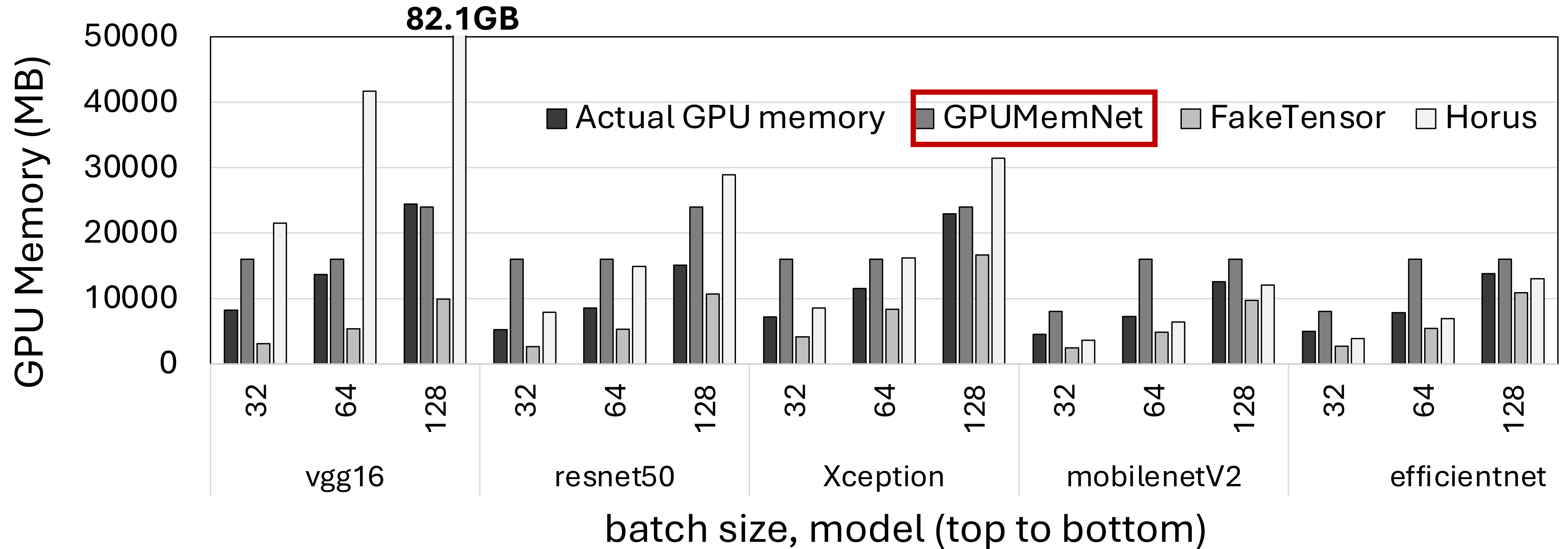


https://github.com/itu-rad/GPUMemNet

Results

Dataset	Estimator	Class Range Size	#Classes	Accuracy
MLP	MLP	1GB	5	0.95
	MLP	2GB	4	0.97
	Transformer	1GB	5	0.97
	Transformer	2GB	4	0.98
CNN	MLP	8GB	6	0.82
	Transformer	8GB	6	0.81
Transformer	MLP	8GB	6	0.87
	Transformer	8GB	6	0.85

Unseen real-world models



GPUMemNet overestimates GPU memory because of how we formulated the problem!
Overestimations take away potentials for collocation!

Summary

- Traditional programming vs. ML-based solutions
- Different types of Machine Learning practices
- Where to train and do inference.
- A real-world problem: GPUs are underutilized and collocation can be an opportunity. GPU memory estimation is needed for robust collocation
- GPUMemNet
  - Dataset
  - Tools for extending the dataset

https://huggingface.co/datasets/ehyo/GPU-Resources-Estimation-for-Deep-Learning-Training-Tasks

https://github.com/itu-rad/GPUMemNet

https://arxiv.org/abs/2602.17817