PhD Course: Optimized Execution of Neural Networks at the Edge

Daniele Jahier Pagliari, Alessio Burrello
daniele.jahier@polito.it, alessio.burrello@polito.it
Department of Control and Computer Engineering, Politecnico di Torino

Politecnico
di Torino

Course Overview

Organization:

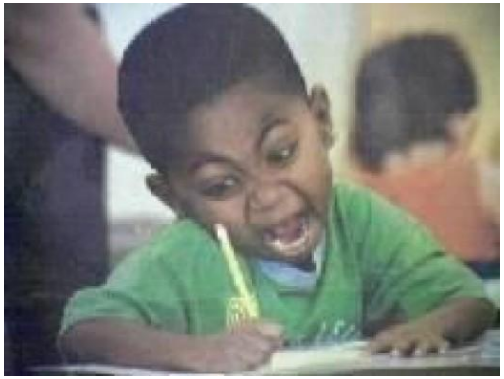
- Duration: 20 hours (hard-skills)

- Schedule:
  - 4 lectures of 4 hours each
  - Final 4h lecture for students' presentations
  - Exact schedule:
    - Tue, June 18th → 9-13, Room **C**
    - Fri, June 21st → 9-13, Room **C**
    - Tue, June 25th → 9-13, Room **C**
    - Fri, June 28th → 9-13, Room **C**
    - Tue, July 2nd → 9-13, Room **C** (Exam)

This one has been moved a bunch of times, sorry for that

Course Overview

- **Exam:**



1. Select one (or more) paper that touches on one of the course's subjects
   - Select autonomously, or ask the instructors if you have no ideas

2. Make a flash presentation (3 min) of the paper to the rest of the class during the final lecture
   - Pass with Merit: show how the paper's content could be used for your own research.

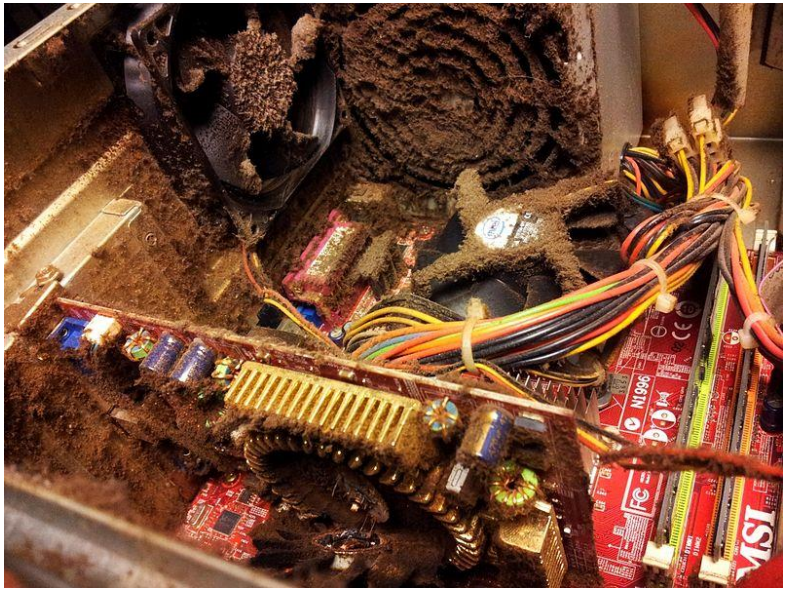
Aim of the Course

- Justify **why we need to run neural networks efficiently** at the edge
  - … and not only at the edge.

- Describe some **state-of-the-art optimization techniques** to improve NN efficiency.
  - …and/or to trade-off efficiency for accuracy

Aim of the Course

- Beware: we might get dirty…
  - This is not a HW design course, but we'll need some HW knowledge to explain some of the concepts



Outline of the Course

1. Introduction

2. Efficiency-oriented Optimizations for DNNs
   - Neural Architecture Search
   - Pruning
   - Quantization
   - Adaptive/dynamic models

3. DNN Compilers

4. **Hands-on Examples in Python**

Me

Prof. Burrello

Your Background

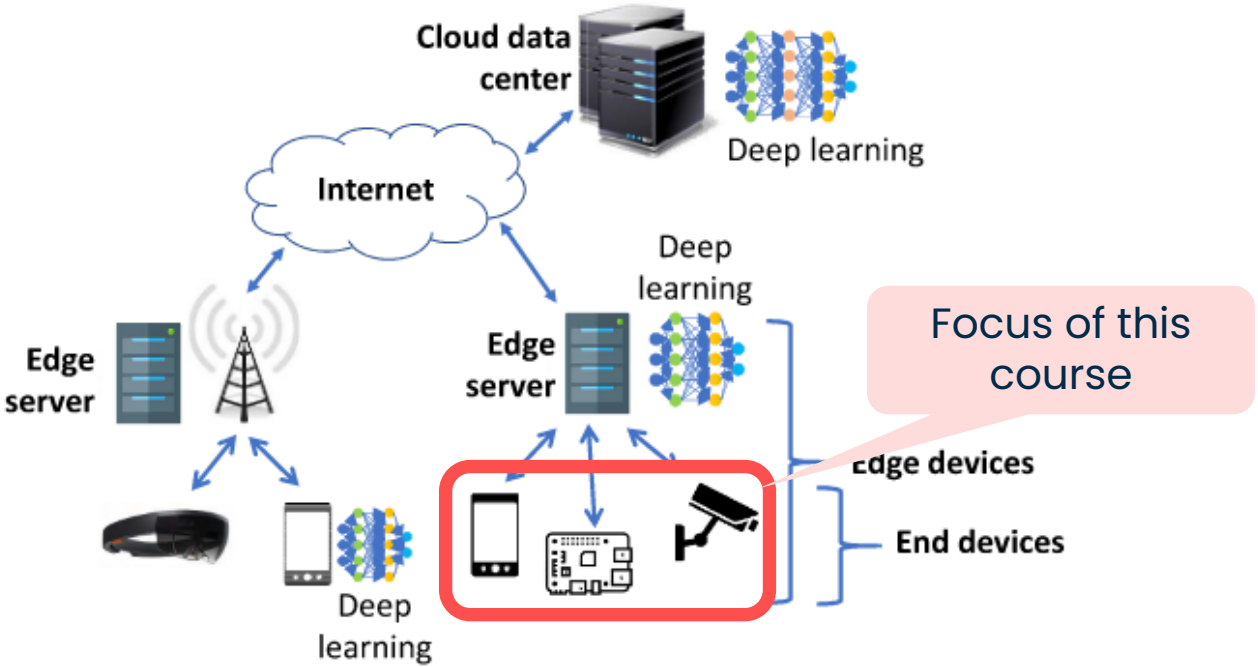
- Where do you come from?
  - PhD Major
  - Reserch Field

Politecnico
di Torino

Motivation

AI in the Internet of Things (IoT)

- Where does AI run?

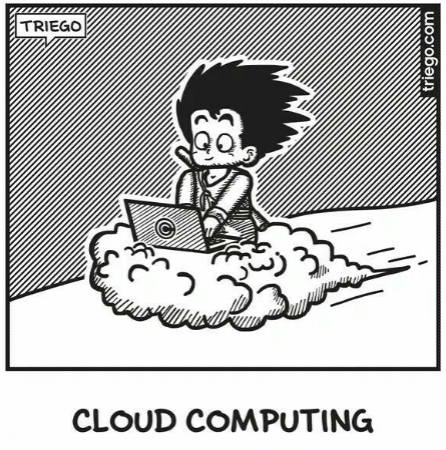


[Source] J. Chen and X. Ran, "Deep Learning With Edge Computing: A Review," Proceedings of the IEEE, vol. 107, no. 8, pp. 1655–1674, 2019, doi: 10.1109/JPROC.2019.2921977.

Why Run Neural Networks at the Edge

- Machine (and Deep) Learning models are at the core of many emerging applications in the IoT world:
  - Smart city/district/factory/building/etc.
  - Bio-signals processing
  - Context-aware interactions (activity tracking, location-awareness)
  - Etc.

- **But why don't we just run everything in the cloud?**



Cloud Computing is not Always Optimal

- **Reason #1: Network Pressure**

- **Example:**
  - Classify 224x224 video frames from IoT cameras in real time at 30fps.
  - Must transmit 224x224x3 bytes to the cloud in 30ms
    - + 1 byte for the class index, negligible

  - Upload bandwidth for single camera = (224x224x3x8)/0.03 = **40Mbps**

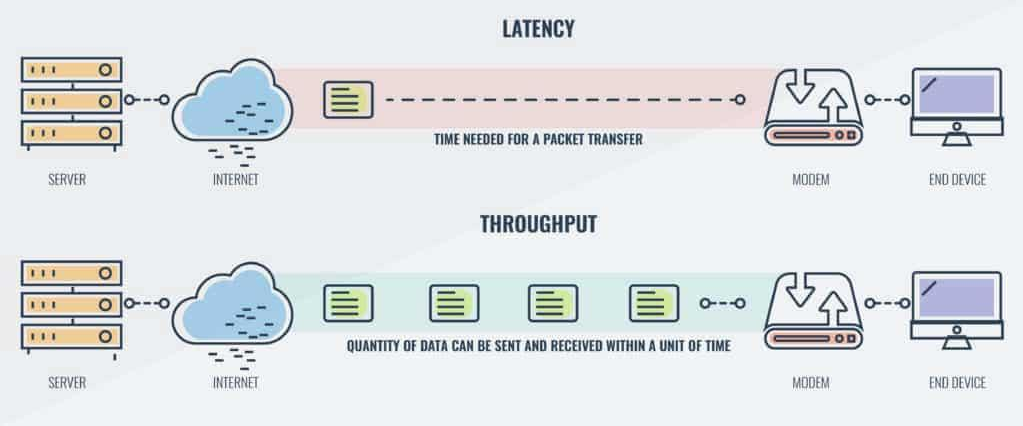
  - This is a **very low resolution** for todays' standards
  - A real system could have **100s of cameras** in the same network.

Cloud Computing is not Always Optimal

- **Reason #2: High and unpredictable latency (low responsiveness)**
  - Connected with the previous one, but distinct.
  - Wireless WAN links (4G, LoRA, etc.) have significant **Round-Trip Times** (RTTs). Even if the bandwidth is high, the time to get the first packet back is 10-100 of ms.
  - Wireless connections can be unstable or unavailable in some places.

Bandwidth vs Latency

- The bandwidth issue is related to **scale** (it happens because you have 1000s or millions of connected devices)
- The latency problem is there even for a **single** device.



Cloud Computing is not Always Optimal

- **Reason #3: High energy consumption**

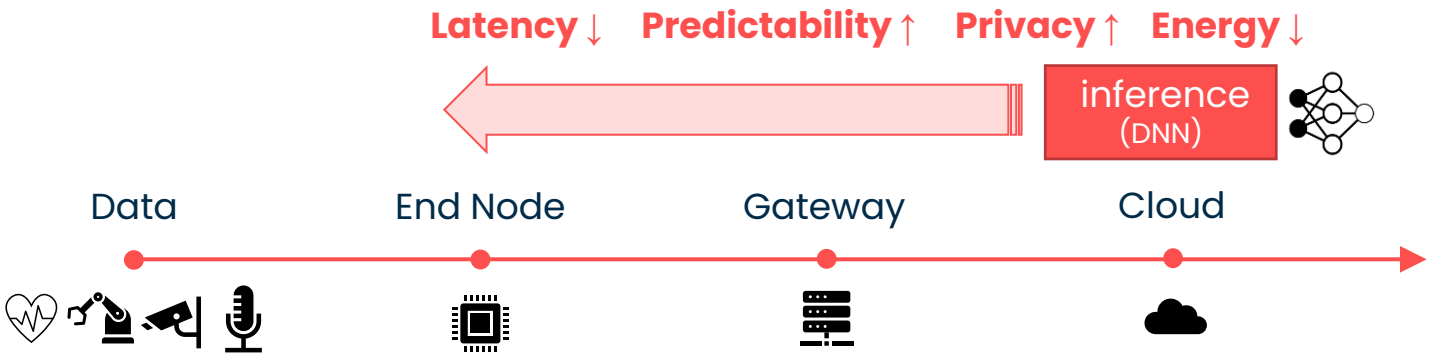
  - **Computation energy** is constantly decreasing
    - Technology scaling, new types of devices, architectural improvements, software improvements
  - **Wireless transmission energy** is not decreasing at the same pace
  - **Solution**: do at least part of the computation locally and trasnmit aggregated information (sort of compression).

Cloud Computing is not Always Optimal

- **Reason #4: Security and Privacy**
  - Transmitting sensitive raw data to the cloud raises privacy concerns.
  - Users must fully trust cloud providers for ensuring that their private data is stored and processed in a secure way.
  - Aggregate information might be less sensitive.
    - E.g. a label saying “there’s a person” versus a photo of the person

Summary

- Near-sensor DNN inference has several potential benefits w.r.t. a traditional cloud-centric approach:
  1. Less network pressure
  2. More predictable and lower latency
  3. Lower energy consumption
  4. Data privacy



Ok, So Let's Do Everything at the Edge

- Not so fast…

- Different definitions of "edge"…
  1. **Edge "gateway":** embedded SoC (Cortex-A class)
  2. **End-node (sensor):** typically microcontroller-based (Cortex-M class)

- In both cases, the **hardware specs differ by orders of magnitude** from those available in the cloud.

Edge AI

- Large AI models (especially DNNs) simply cannot run “as is”:

Hardware Metric	Data Center HW (NVIDIA DGX Station)
N. of compute units (cores)	20 (CPU cores) 20000 (CUDA cores) 2560 (Tensor Cores)
Clock Freq.	2.2 GHz (CPU)
Memory (main)	256 GB (Main) 128 GB (GPU)
Memory (mass)	8 TBs (SSD)
Peak Power consumption	≈ 1500 W
Cost	50000 €

*** Values significantly larger than a “typical” edge device, since GAP9 is specialized for DSP/AI workloads

Definitely not fitting in GAP9’s main memory!

AC/cycle
video!

≈ 50s/inference @ 10 MAC/cycle on GAP9 → Not OK for…. **anything!!**

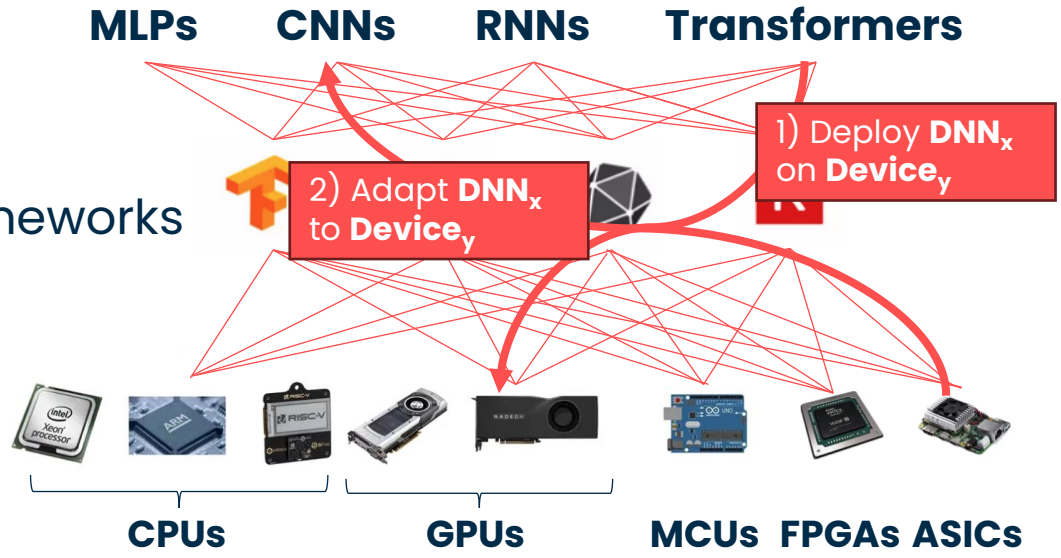
Model (on ImageNet)	Parameters (Memory)	FLOps
ResNet-50	**25 M (100 MB)**	**8**
ViT-L	**305 M (1.2 GB)**	**191.2**

TinyML Issues

- We **just cannot** deploy the same models designed for the cloud…
- Also, there's a plethora of different edge devices, ML frameworks, and problems to be solved (the **deployment challenge**):
  - Every (task, device) needs a different, highly-optimized, **tiny DNN model**
  - Huge space to explore by hand. **Automation** is the key.

Automating AI Optimization and Deployment

- Heterogeneous DNNs
- Heterogeneous SW Frameworks
- Heterogeneous HW



Summary

- Running NN-based inference (…or training) at the edge may provide benefits in terms of **latency, scalability, energy, and privacy**.

- But doing so is not easy due to the limited hardware capabilities of these devices, mainly determined by **cost** and **energy efficiency** reasons.

- So, we need specific techniques to optimize the inference **memory occupation**, **latency, throughput** and **energy consumption** on cost- and energy-constrained IoT devices.

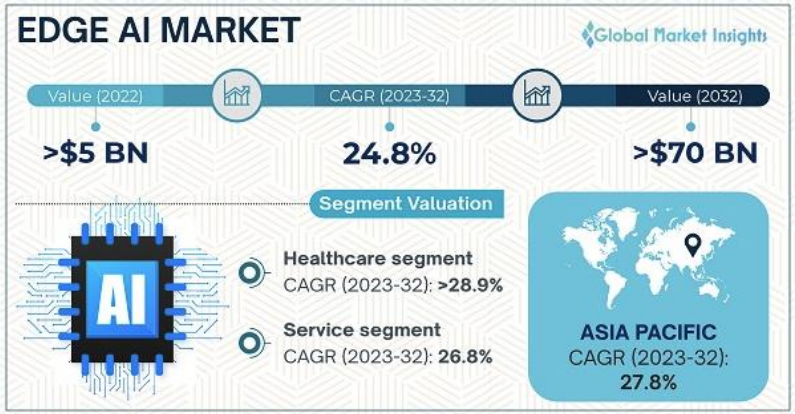
Course Focus

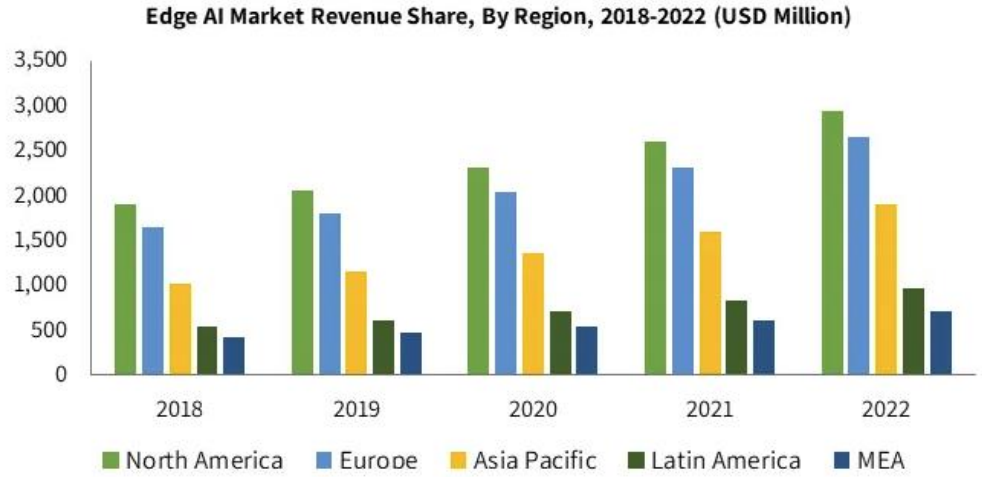
- This course focuses mostly on **general purpose** HW
  - CPUs, Microcontrollers
  - Most techniques apply also to domain-specific HW (GPUs, TPUs)
  - If we have time, we'll make a brief parenthesis on NN accelerators…
- We also focus mainly on **inference:**
  - We don't analyze optimizations for **on-device** training
  - **Don't be confused**: optimizations for inference do start during training!

However…

- Cloud devices are powerful, but serve millions of requests.
- Many of the optimizations analyzed here are beneficial also to make cloud inference lighter
  - Faster, more energy efficienct, etc.
- Some of them can also be used to speed-up training…

A Relevant Business

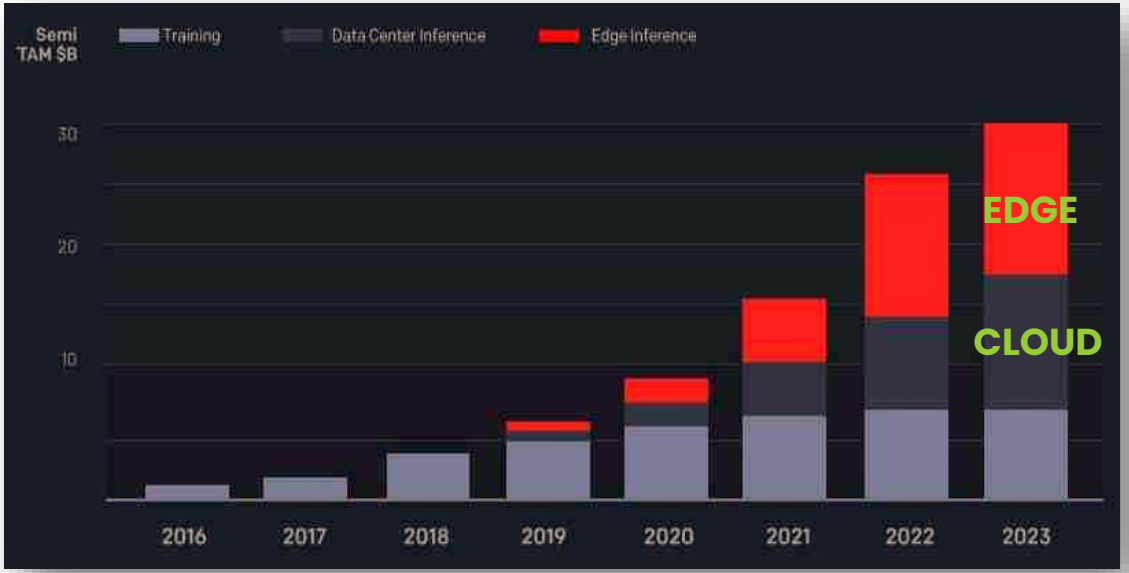




[Source] https://www.gminsights.com/industry-analysis/edge-ai-market

A Relevant Business

Semiconductor Total Available Market of AI workloads



[Source] Barclays research

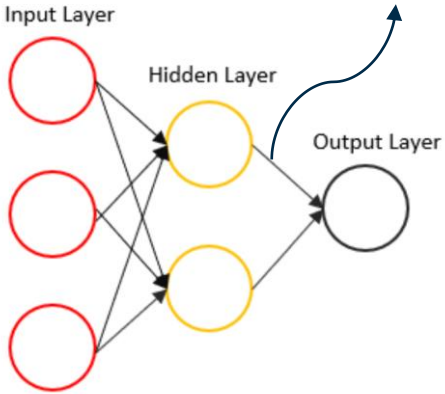
Politecnico
di Torino

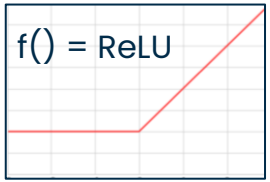
DNN (Inference) from a Computational Viewpoint

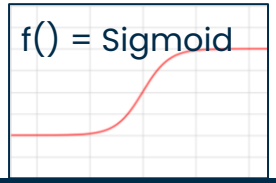
Background: DNNs

- Model of computation *roughly* inspired by the human brain.
- Stacking of multiple layers of artificial neurons

$$y_j^{(l)} = f(\sum w_i^{(l)} x_i^{(l-1)} + b_j^{(l)})$$







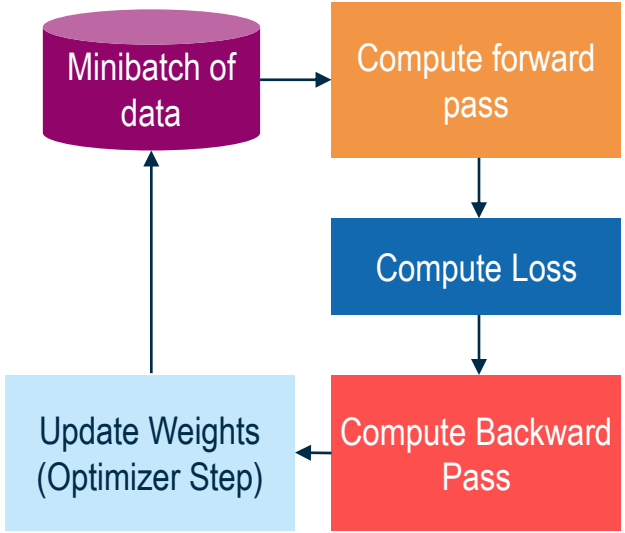

```python
# pytorch definition
import torch
import torch.nn as nn
import torch.nn.functional as F

class Net(nn.Module):
        def __init__(self):
                super(Net, self).__init__()
                self.l1 = nn.Linear(3, 2)
                self.l2 = nn.Linear(2, 1)

        def forward(self, x):
                y1 = F.relu(self.l1(x))
                y2 = F.sigmoid(self.l2(y1))
                return y2
```

Background: DNNs

- Trained by back-propagating gradients of an error (loss) function



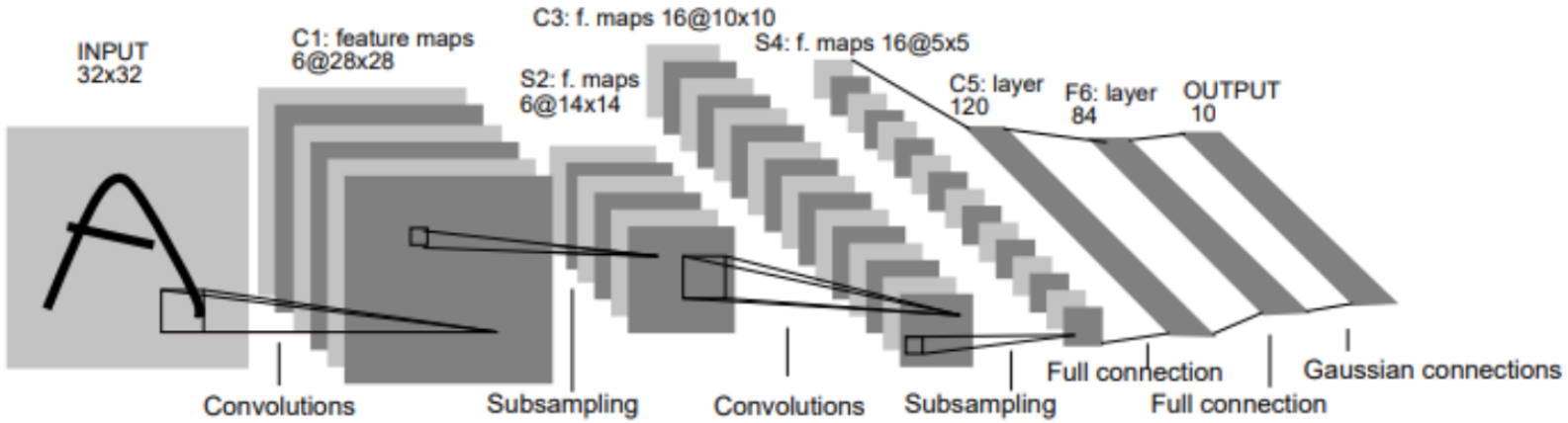

```
model = Net()
loss = nn.BCELoss()
optimizer = torch.optim.SGD(
        model.parameters(), lr=0.001)

for epoch in range(EPOCHS):
  model.train(True)
  for minibatch in training_set:
    inputs, labels = minibatch
    optimizer.zero_grad()
    outputs = model(inputs)
    loss = loss_fn(outputs, labels)
    loss.backward()
    optimizer.step()
    # optional logging, etc.

  # eval phase on validation set…
```

Background: Deep Neural Networks

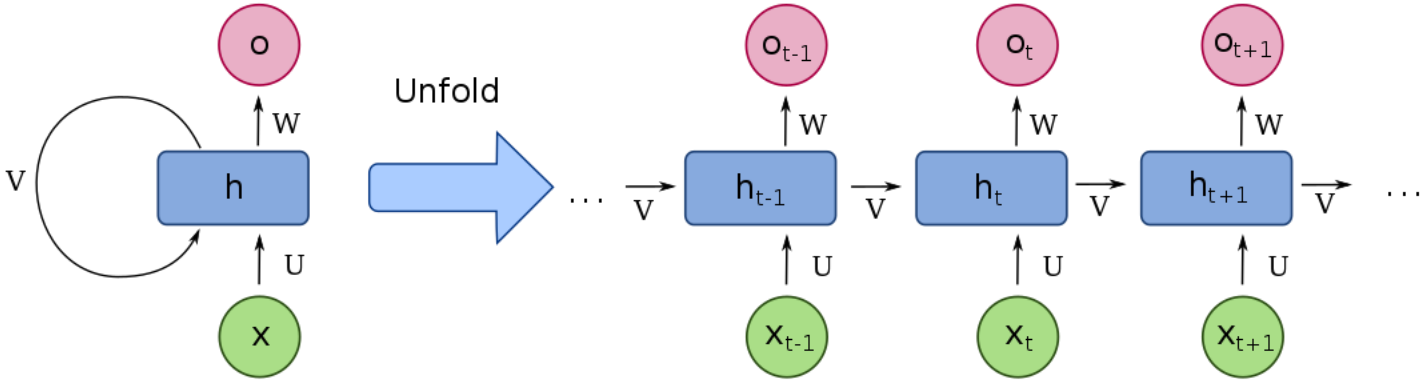
- Convolutional Neural Networks (CNN) for Computer Vision



Source: LeNet-5

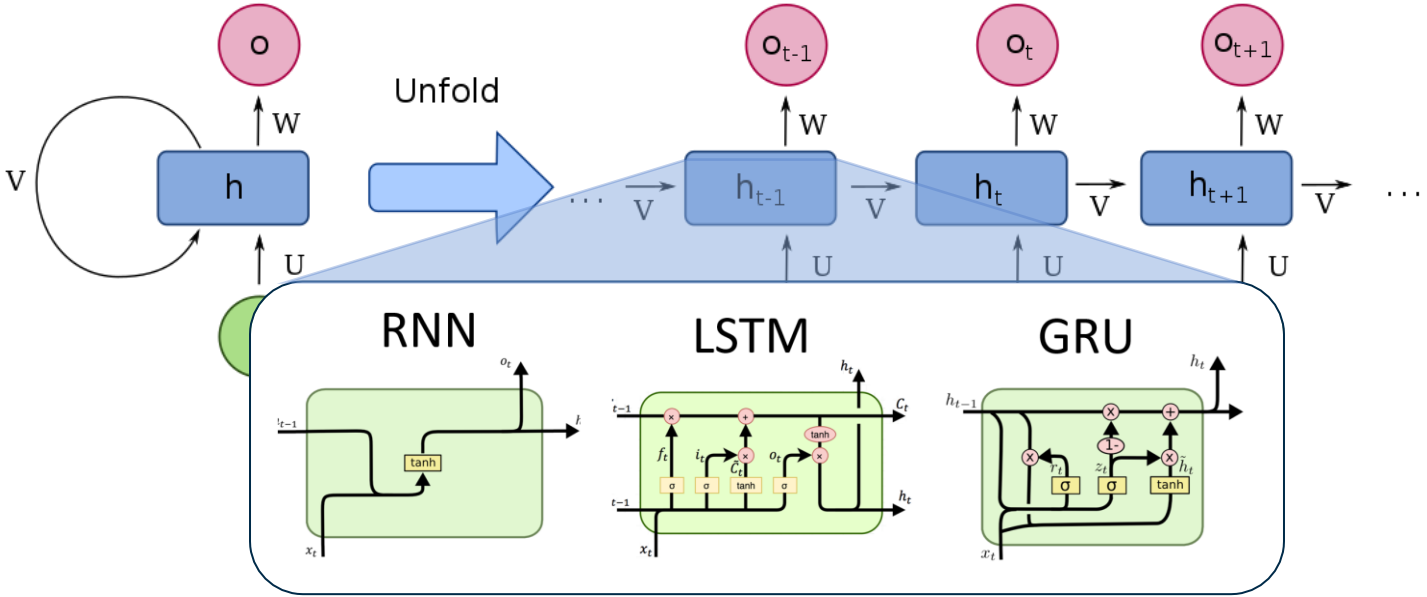
Background: Deep Neural Networks

- Recurrent Neural Networks (RNN) for Sequence Processing



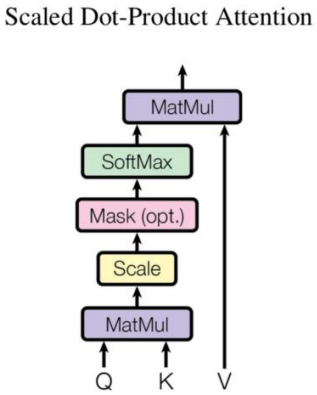
Background: Deep Neural Networks

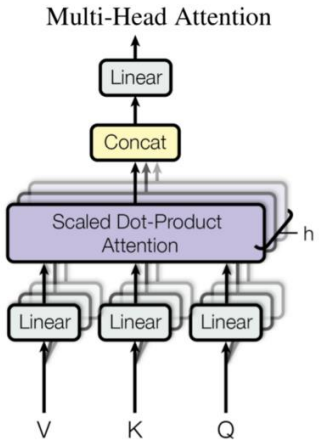
- Recurrent Neural Networks (RNN) for Sequence Processing

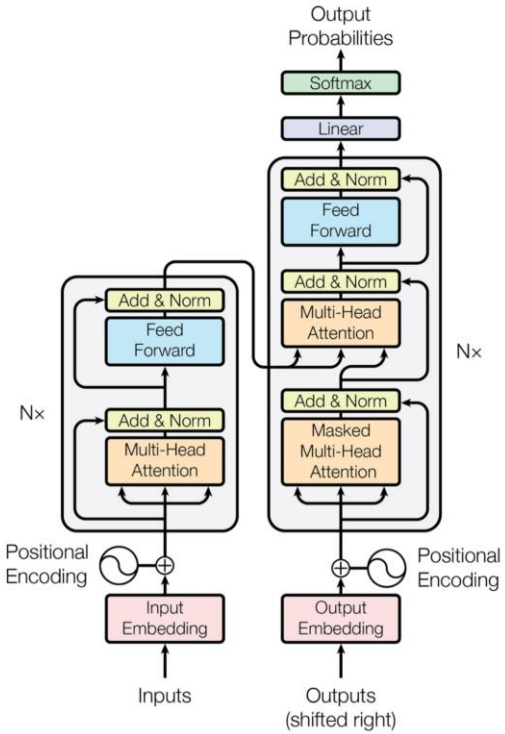


Background: Deep Neural Networks

- Transformers for…. Everything?







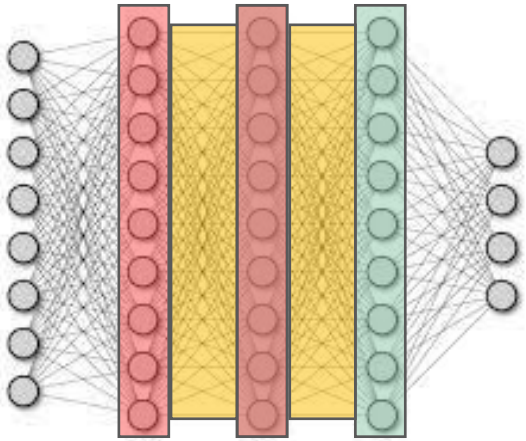
Source: medium

Critical Computations

The critical computations in most NN layers are Matrix-Vector (MxV) or Matrix-Matrix (MxM) products (GEMMs)

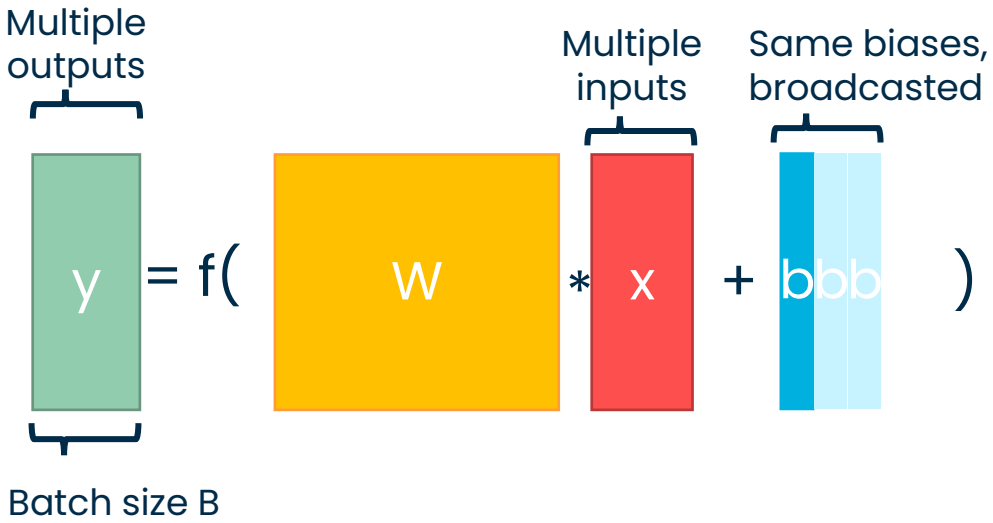
Linear Layers

- Linear, or Fully-Connected Layers:
  - $\mathbf{y} = f(\mathbf{W} \cdot x + \boldsymbol{b})$
  - Key operation is a dense Matrix-Vector (MxV) multiplication



Linear Layers

- When processing a (mini-)batch of N inputs, the computation becomes a MxM:



RNN Cells

- GEMM is also the key operation in RNN cells (e.g. LSTM):
  - The others are element-wise activation functions and gating

$$i_t = sigmoid(W_{xi}x_t + W_{hi}h_{t-1} + b_i)$$

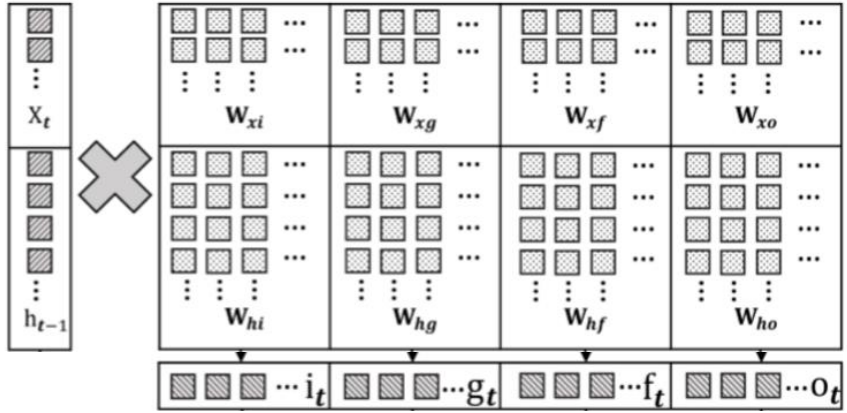
$$f_t = sigmoid(W_{xf}x_t + W_{hf}h_{t-1} + b_f)$$

$$o_t = sigmoid(W_{xo}x_t + W_{ho}h_{t-1} + b_o)$$

$$g_t = tanh(W_{xg}x_t + W_{hg}h_{t-1} + b_g)$$

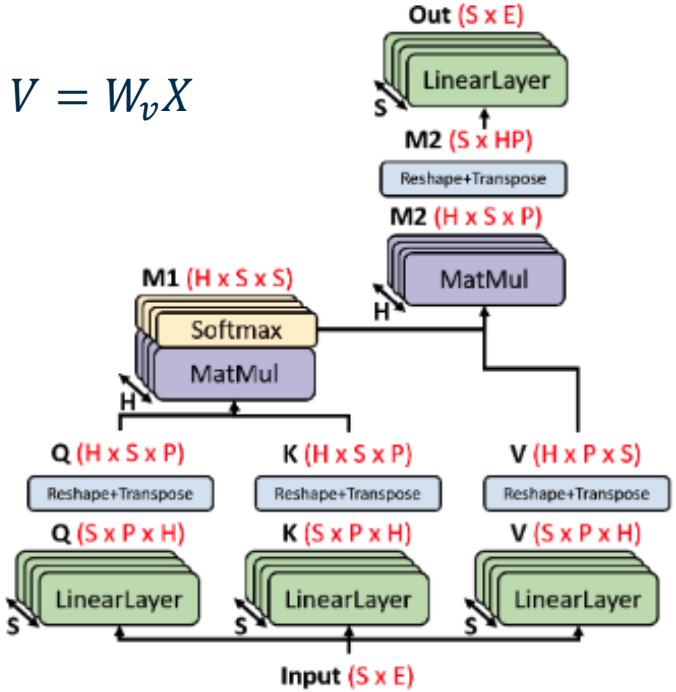
$$c_t = f_t \odot c_{t-1} + i_t \odot g_t$$

$$h_t = tanh(c_t) \odot o_t$$



Attention Layers

- …and in Transformers' Multi-Head Self-Attention (MHSA).
  - **Input Projections:** $Q = W_q X, \quad K = W_k X, \quad V = W_v X$
    - Three linear layers
  - **Attention:** $A = \text{softmax}\left(\frac{QK^T}{\sqrt{P}}\right)V$
    - Two activations-to-activations MatMuls
  - **Output Projection:** $Y = W_y A$
    - Another linear layer

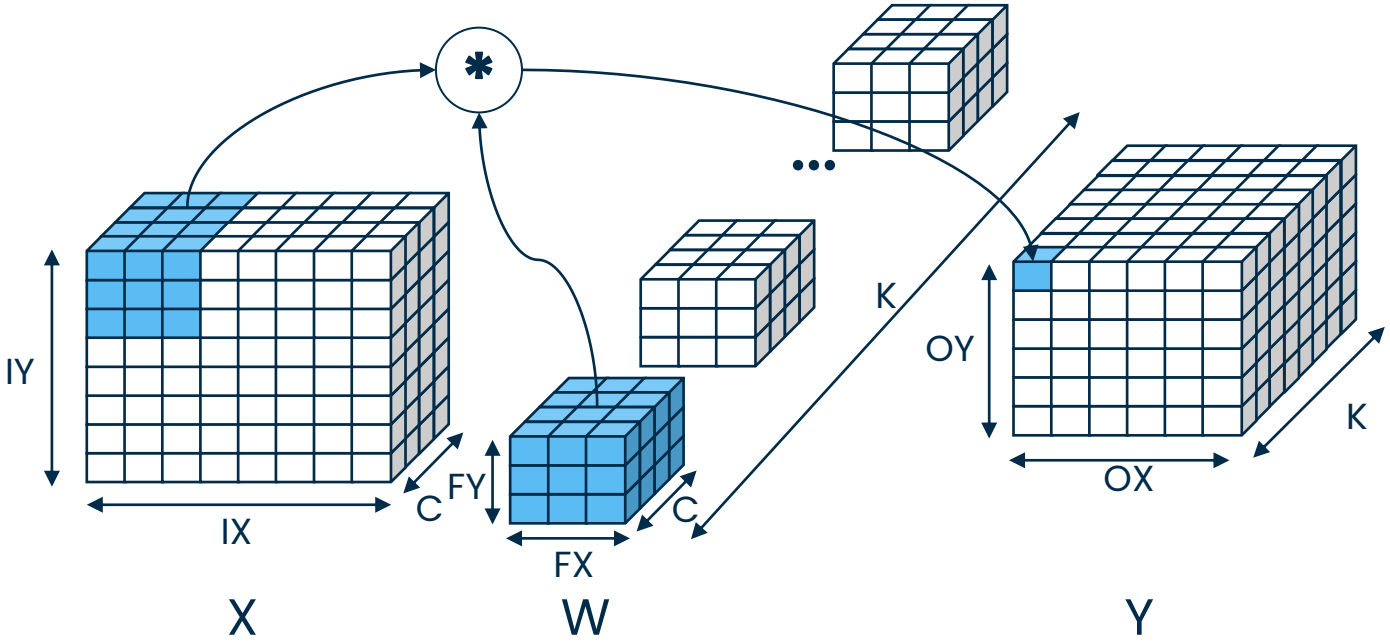


S = sequence length
E = embedding dimension (n. of features).
H = num. of heads
P = projection (hidden) size

What About Convolutions?

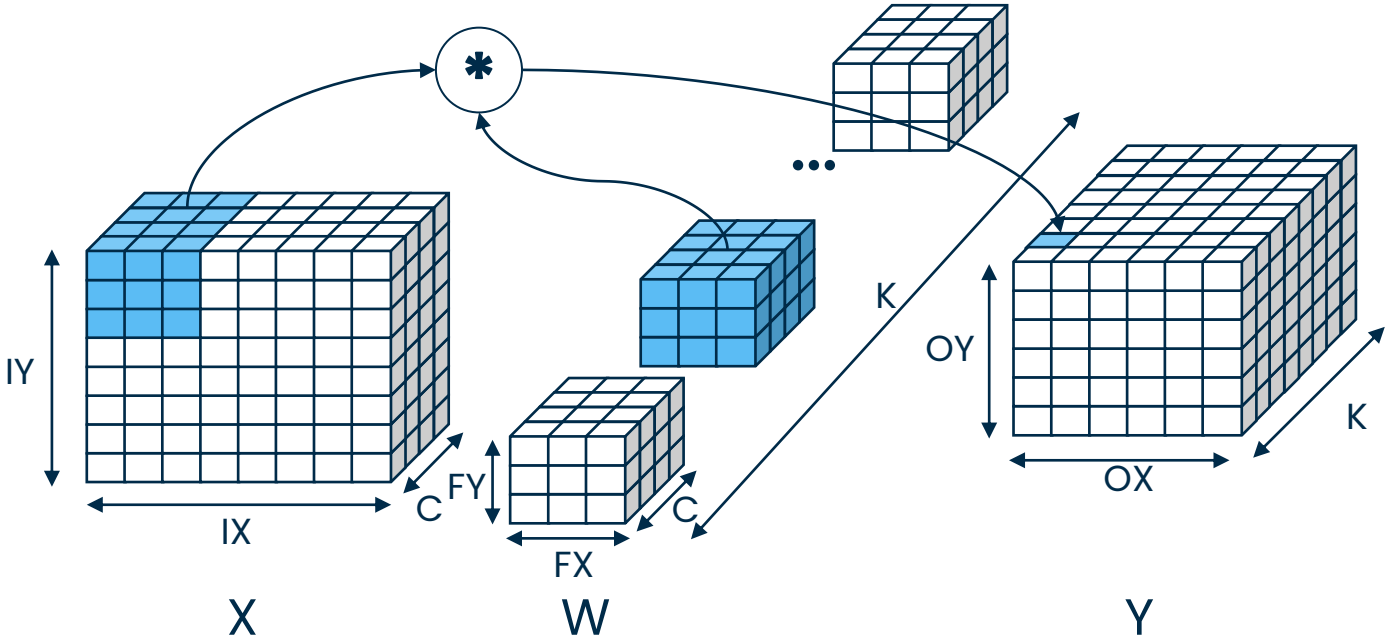
(2D) Convolutions

- $Y_{x,y,k} = f(\sum_{fx=0}^{FX-1} \sum_{fy=0}^{FY-1} \sum_{c=0}^{C-1} W_{k,fx,fy,c} * X_{x+fx,y+fy,c} + b_k)$
- $\forall\, k \in [0, K-1], \forall\, x \in [0, OX-1], \forall\, y \in [0, OY-1]$



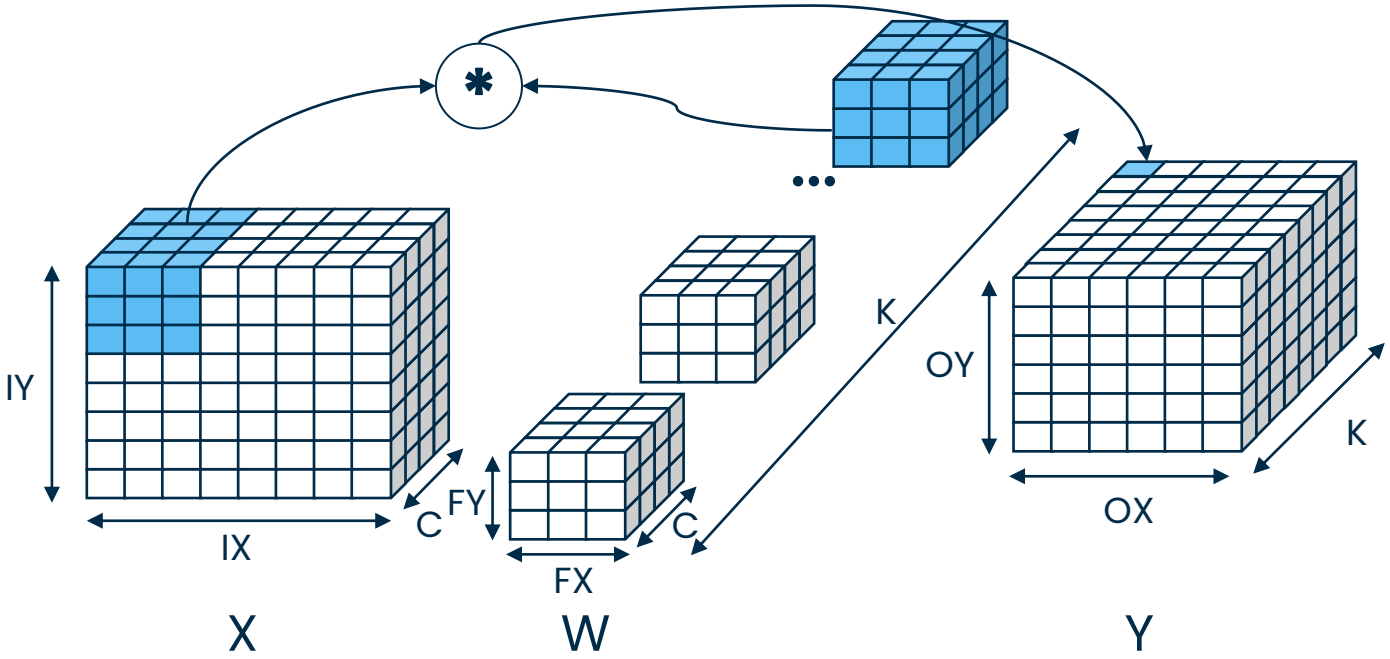
(2D) Convolutions

- $Y_{x,y,k} = f(\sum_{fx=0}^{FX-1} \sum_{fy=0}^{FY-1} \sum_{c=0}^{C-1} W_{k,fx,fy,c} * X_{x+fx,y+fy,c} + b_k)$
- $\forall\, k \in [0, K-1], \forall\, x \in [0, OX-1], \forall\, y \in [0, OY-1]$



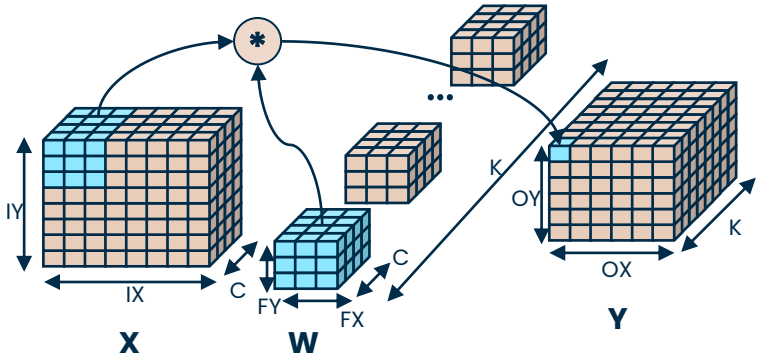
(2D) Convolutions

- $Y_{k,x,y} = f(\sum_{fx=0}^{FX-1} \sum_{fy=0}^{FY-1} \sum_{c=0}^{C-1} W_{k,fx,fy,c} * X_{x+fx,y+fy,c} + b_k)$
- $\forall\, k \in [0, K-1], \forall\, x \in [0, OX-1], \forall\, y \in [0, OY-1]$



(2D) Convolutions

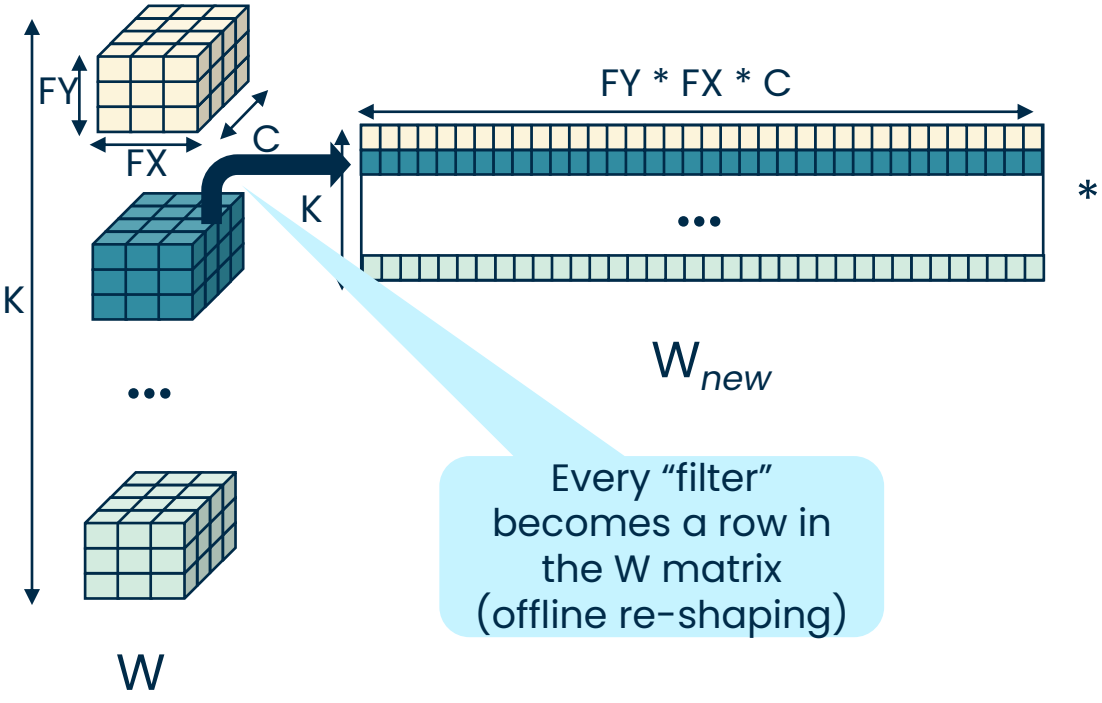
- Naïve implementation requires 6 nested loops:
  - Plus a 7-th if processing a batch of inputs

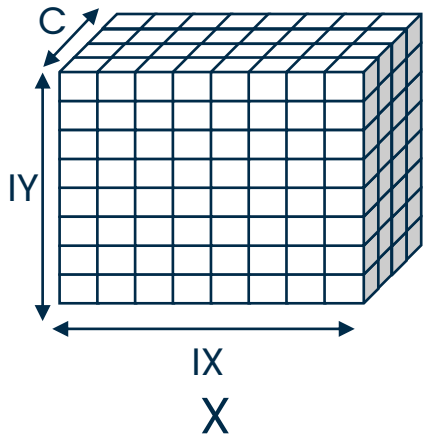


- But Convolutions can be turned into GEMMs too:
  - Through a reshaping process called im2col

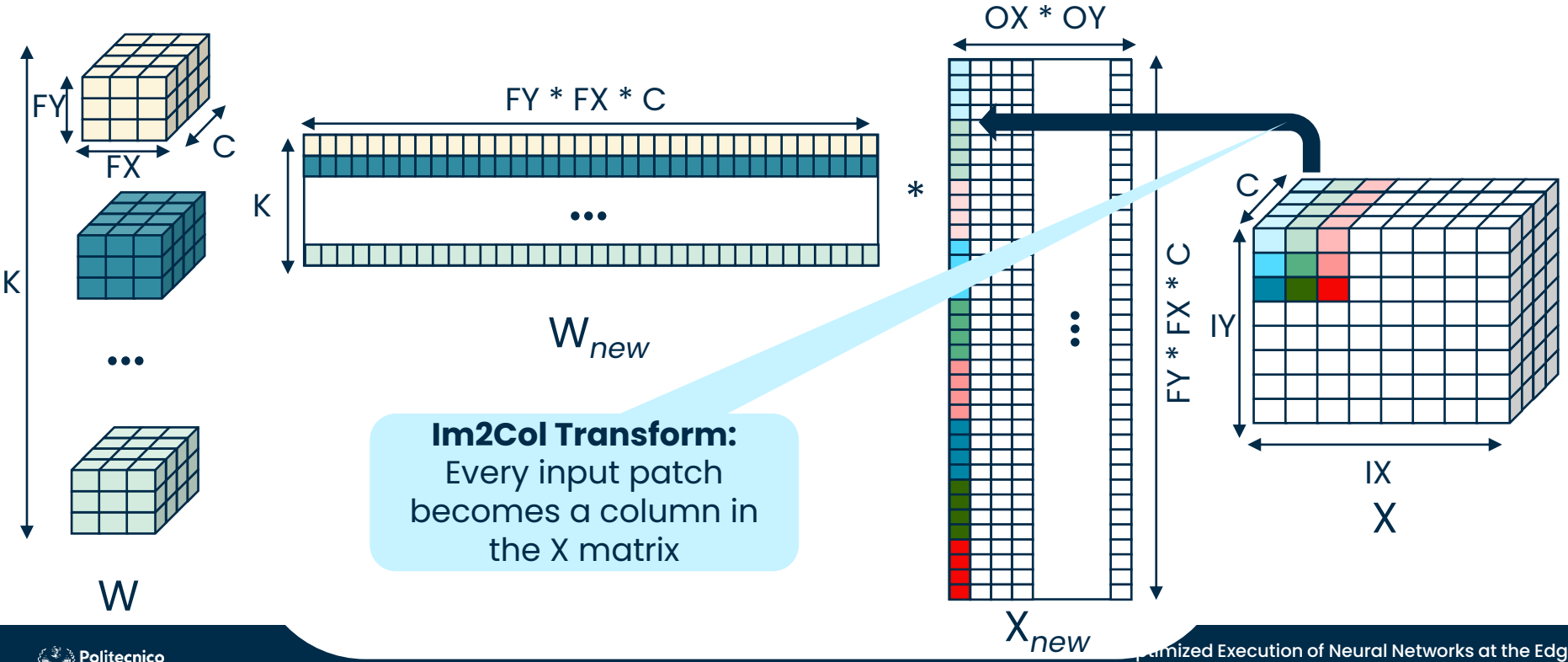
```
for k in range(K):
  for x in range(OX):
    for y in range(OY):
      y[x,y,k] = b[k]
      for fx in range(FX):
        for fy in range(FY):
          for c in range(C):
            y[x,y,k] += (
              W[k,fx,fy,c] *
              X[x+fx,y+fy,c] +
            )
```

(2D) Convolutions as Im2col + MatMul



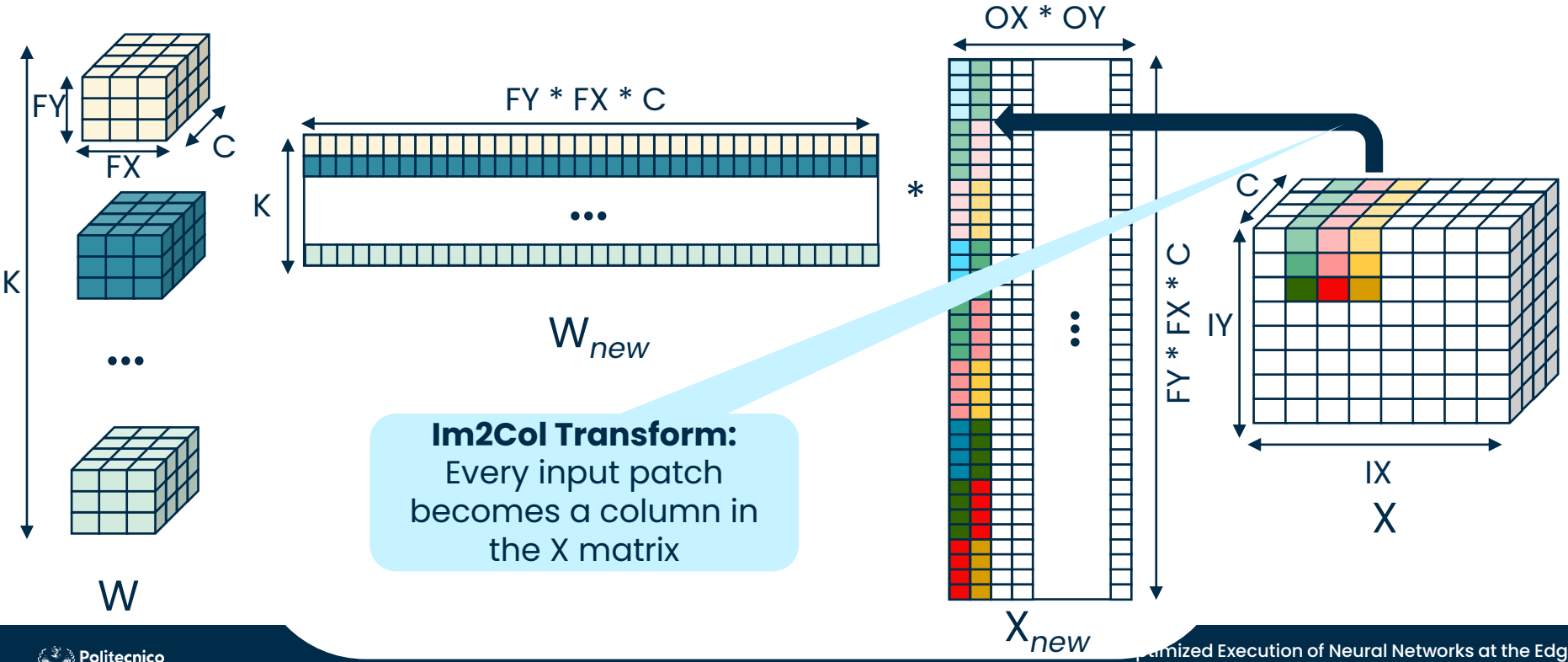


(2D) Convolutions as Im2col + MatMul



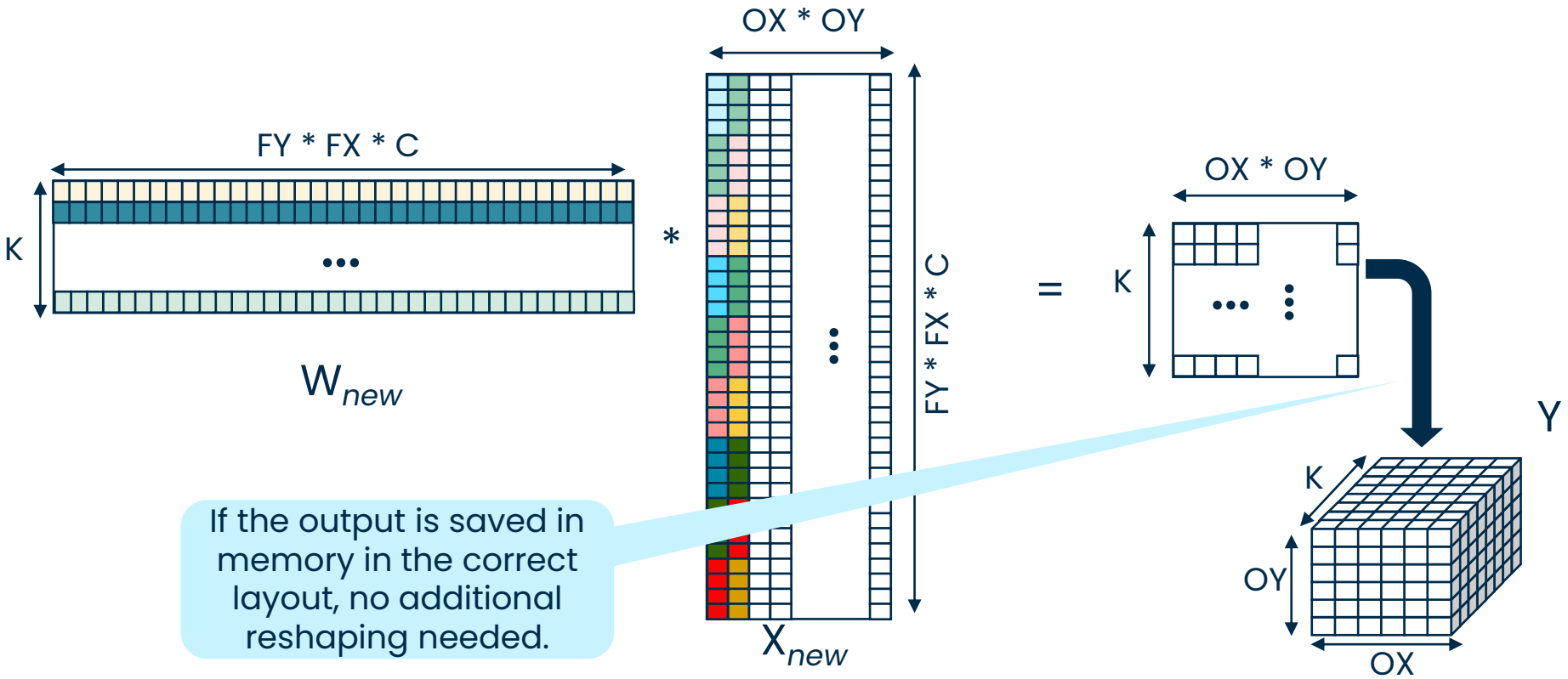
Im2Col Transform:
Every input patch becomes a column in the X matrix

(2D) Convolutions as Im2col + MatMul



Im2Col Transform:
Every input patch becomes a column in the X matrix

(2D) Convolutions as Im2col + MatMul



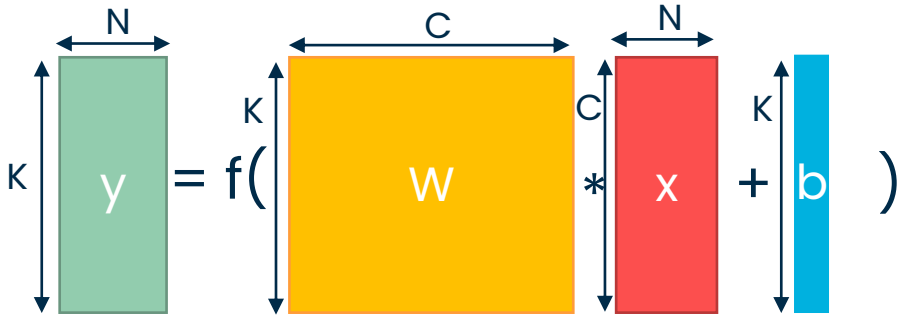
If the output is saved in memory in the correct layout, no additional reshaping needed.

Summary

- Critical NN layers are basically GEMMs plus some data marshalling
  - By the way, that's why GPUs are so good for deep learning….
- If we want to make our models more efficient, we have to optimize the memory occupation, latency, throughput, and/or energy required to perform these GEMMs
- **How can we go about it?**

Complexity

- Let's define generically:
  - X as being of size (NxC), with C the number of ***input features*** or ***channels***
  - Y as being of size (NxK), with K the number of ***output features*** or ***channels***
- Neglecting bias addition and f() application, a GEMM-based layer requires:
  - $N * K * C$ Multiply-and-Accumulate (MAC) Operations, i.e., $\boldsymbol{O(N * K * C)}$ **FLOPs**
  - $\boldsymbol{O(K * C)}$ bytes of non-volatile memory for *weights and biases*
  - $\boldsymbol{O((K + C) * N)}$ bytes of volatile memory for input/outputs (*activations*)



Optimizing DNNs

- **Extremizing, the optimization of DNNs from a computational stand-point can be summarized as the combination of:**

1. **Finding ways to execute GEMM efficiently:**
   - Data-reuse maximization
   - Parallelization
   - Efficient kernels…

No accuracy impact

2. **Reducing the N. of FLOPs/memory:**
   - Optimize the DNN architecture
   - Reduce data bit-widths
   - Skip (or prune) computations…

Trade-off complexity and accuracy

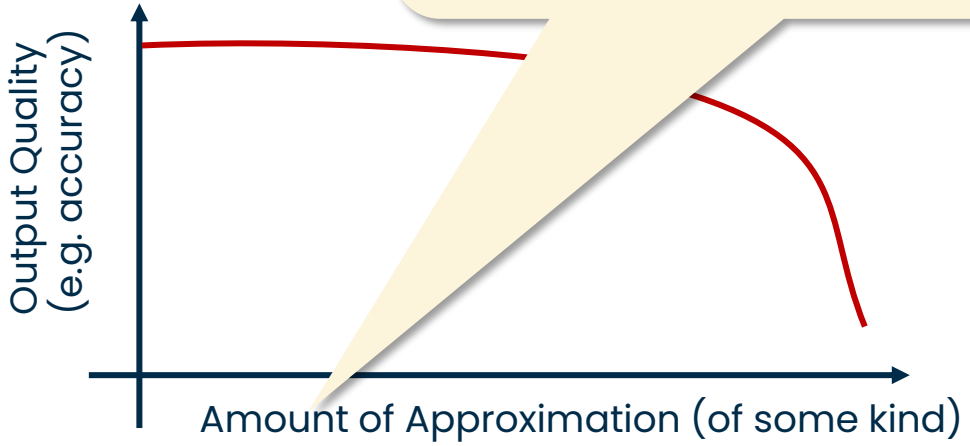
Strategy 1: Executing GEMM Efficiently

- GEMM is arguably the most studied computation pattern ever.
- So, is there **anything to do**, besides selecting highly-optimized GEMM routines (e.g., BLAS)?
- Yes, there is…
  - Leverage specific characteristics of DNN GEMMs (e.g., peculiar and known geometries, known tensor lifetimes, etc.)
  - Combine GEMM and non-GEMM operations to reduce data movement (op. fusion)
  - etc.
- This is handled by so-called **DNN Compilers** and **Backend software libraries**.
  - Overview today, more focus in **Lecture 3**

Strategy 2: Domain-specific Approximations

- Exploit the fact that deep NNs are quite tolerant to approximations.
- **Most potential is here!**

Examples of “approximations”:
- Reduce **n. of computations** (NAS, Pruning)
- Reduce **data precision** (quantization)
- etc.



Why are DNNs Tolerant to Approximations?

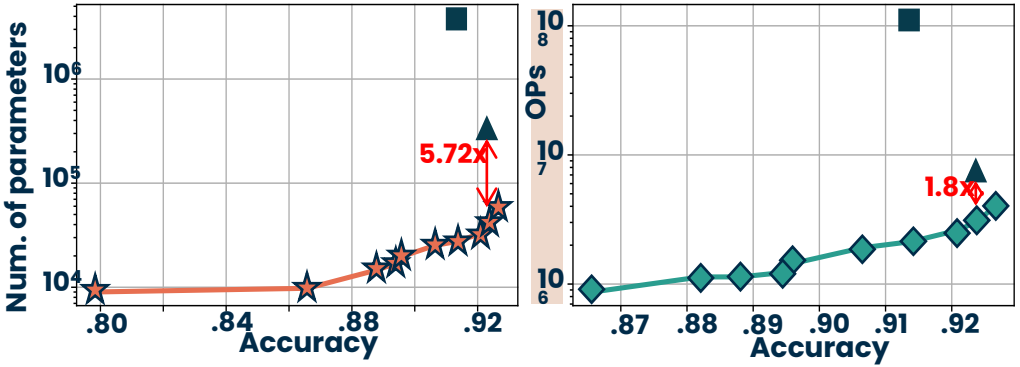
- Exact numerical output not always needed

- Example, softmax-based classifier:

Why are DNNs Tolerant to Approximations?

- DNNs computations are often highly redundant.
  - They use many more parameters and FLOPs than those needed to reach a certain accuracy
    - Because the underlying relation learned by the model is **unknown**
  - Also, most DNNs are typically designed **by hand based on rules of thumb**
    - Many data scientists don't care about complexity.

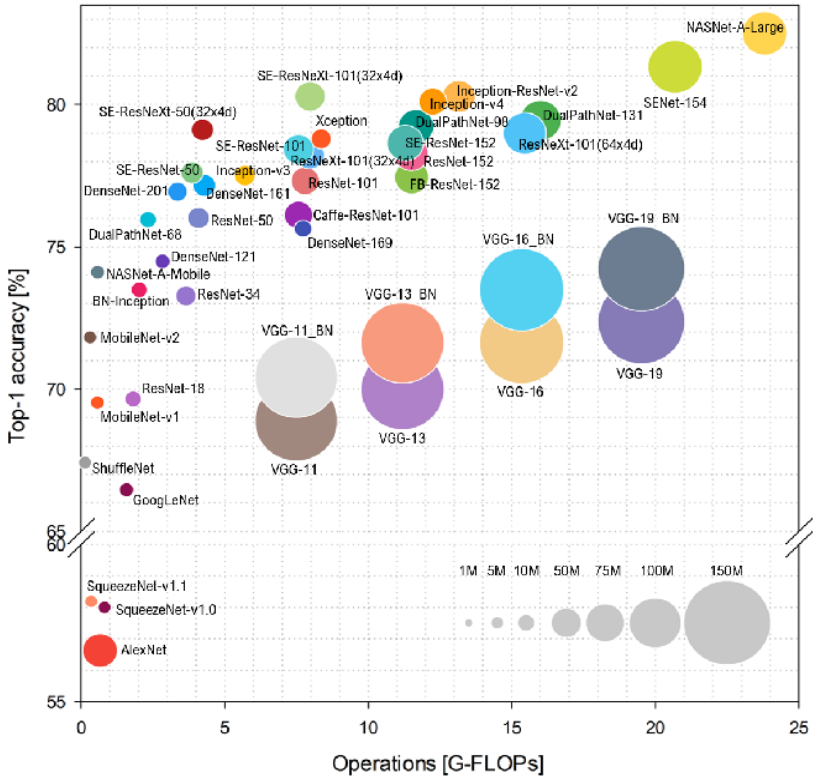
  - Example: Keyword spotting (hand-tuned vs optimized)



[source] M. Risso et al, "Lightweight Neural Architecture Search for Temporal Convolutional Networks at the Edge", IEEE TCOMP 2022

Error Tolerance in DNNs

- Example: the ImageNet landscape
  - Because of this redundancy, the best way to improve the efficiency of a model is often to simplify its architecture!
  - Reduce n. of layers, change type of layer, downscale input, etc.
  - **Anything better than trial-and-error?** We'll see!!



[Source] Benchmark Analysis of Representative Deep Neural Network Archit

Politecnico
di Torino

Strategy 1

Executing MatMul Efficiently

Execute MatMul Efficiently

1. **Parallelize** the computation as much as possible (at multiple levels):
   - In our case: shared-memory multi-core + SIMD parallelism

2. …While trying not to be **memory bound** (**data reuse**):
   - Quite possible, especially for MxV

The Roofline Model

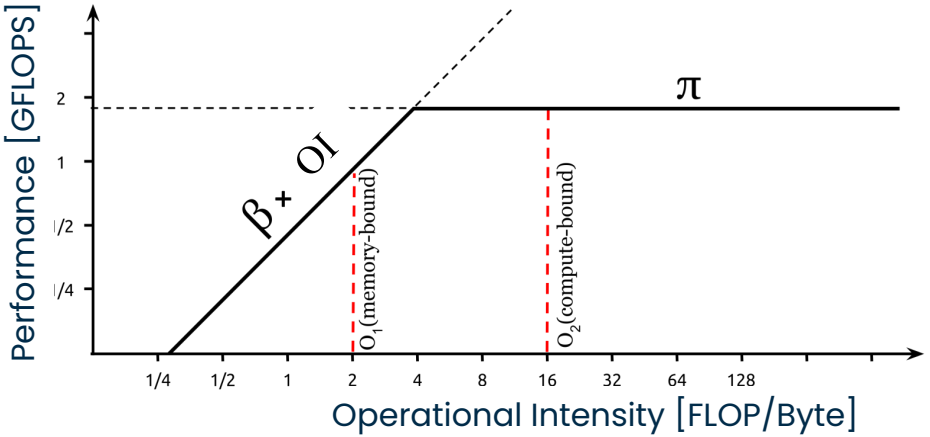
- The performance bound of a computation can be visualized using the "**roofline model**"
  - As a function of its **operational intensity (OI)**
  - ..and of the characteristics of the HW platform.

- The OI is a property of **a given implementation** of a computing task.

$$OI = \frac{N.of\ Operations}{Bytes\ Transferred}\ \left[\frac{FLOP}{Byte}\right]$$

Roofline Model

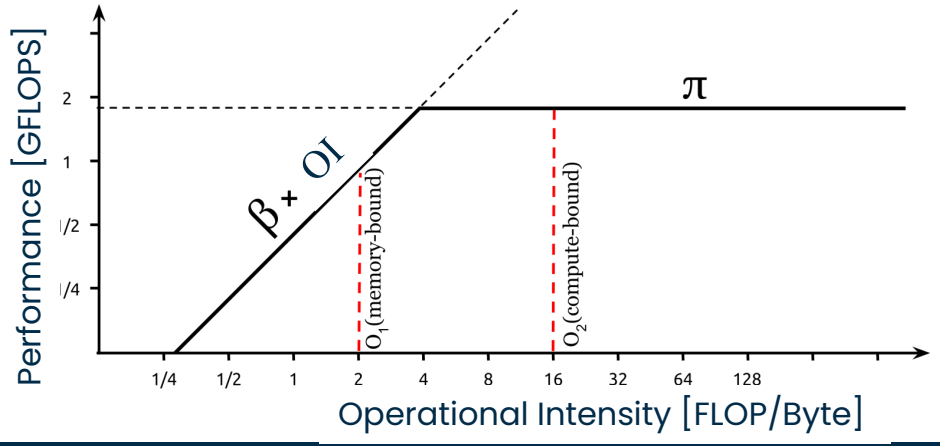
- Two possible upper bounds to the performance of a task for a given HW:
  - **Peak performance**
  - **Peak memory bandwidth.**
- Depending on the OI, we can easily visualize what is the limiting factor for performance, for a particular implementation.



π = peak performance [GFLOPS]
β = peak bandwidth [Byte/s]

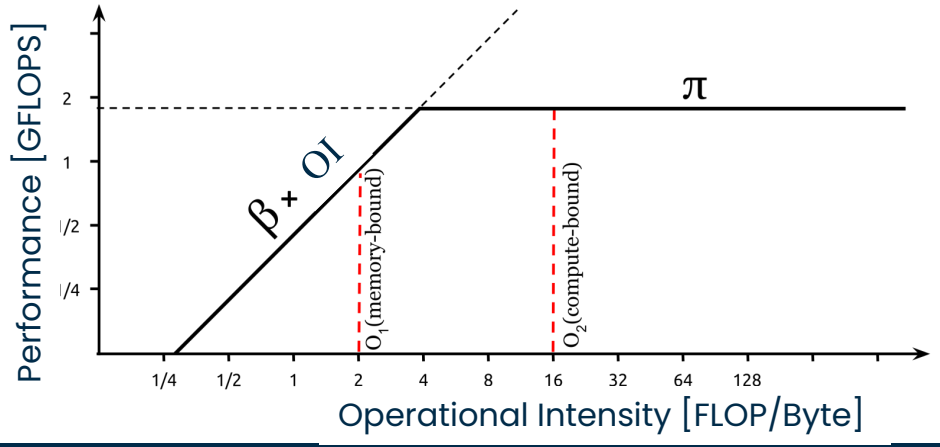
Operational Intensity and the Roofline Model

- **Example:** Assume a system that can perform 2 GFLOPS and can load data from memory at around 0.5GB/s.



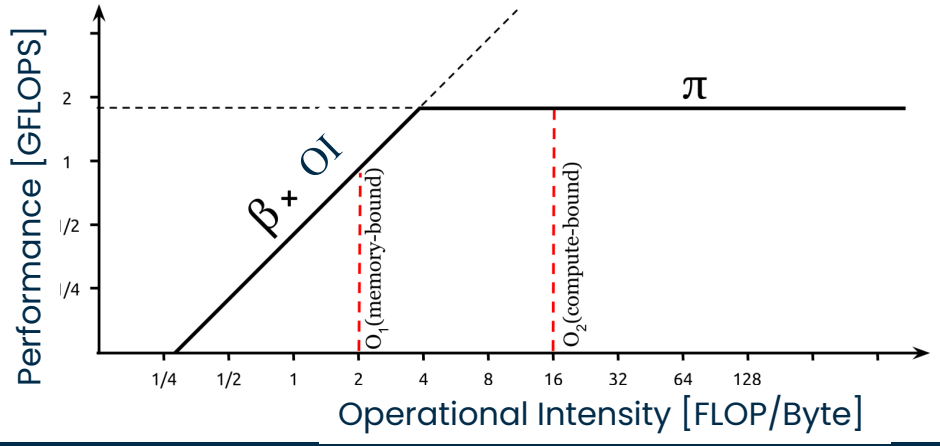
Operational Intensity and the Roofline Model

- **Case 1:** The target computation performs 2 FLOPs for each new loaded byte of data.
- We have OI = O_1 = 2, i.e. the leftmost red line
- We are on the left side of the roofline model, which means that our performance will be limited by memory. Since we can only load data at 0.5GB/s, and we need 1 new byte every 2 operations, we effectively run at max 2*0.5 = 1 GFLOP per second.

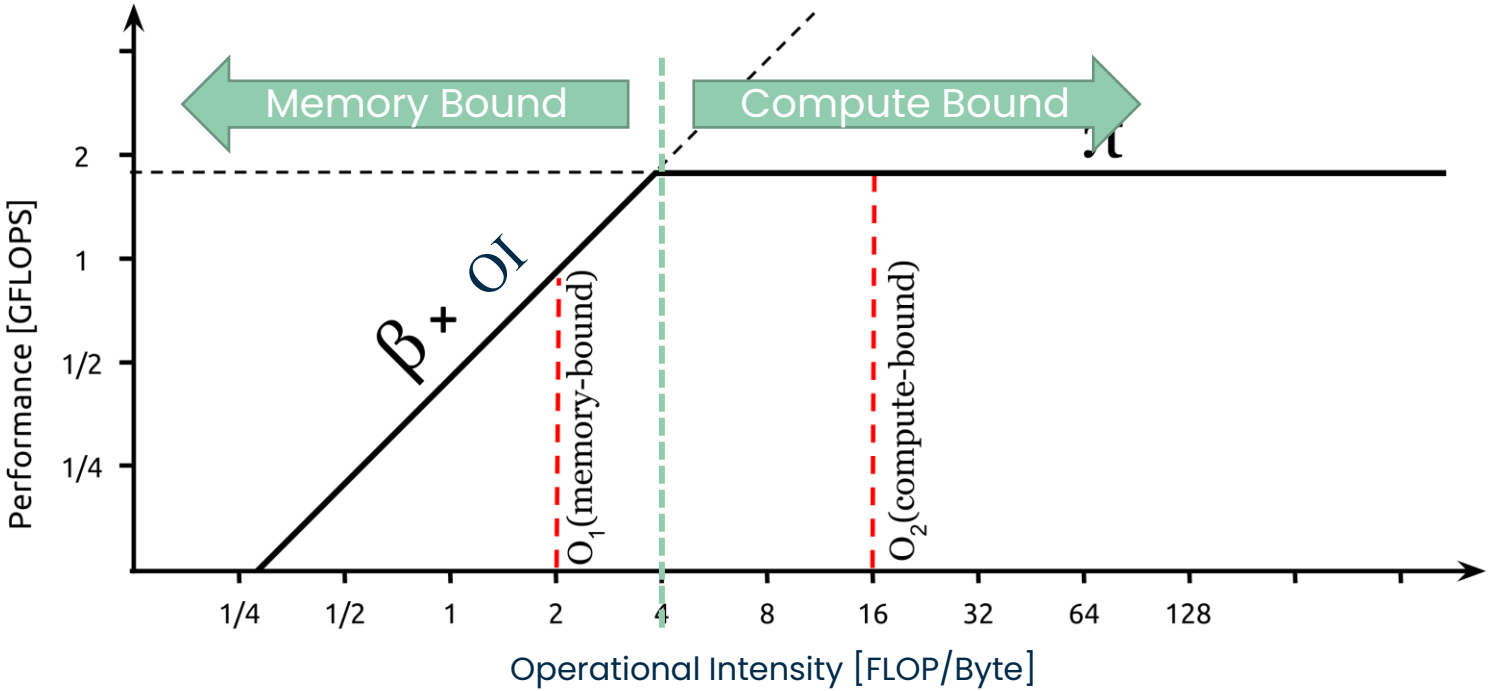


Operational Intensity and the Roofline Model

- **Case 2:** The target computation performs 16 FLOPs for each new loaded byte of data.
- We have OI = O_2 = 16, i.e. the rightmost red line
- We are on the right side of the roofline model, which means that our performance will be limited by compute. Intuitively, we can load all the data needed to keep the ALUs of our system constantly active.



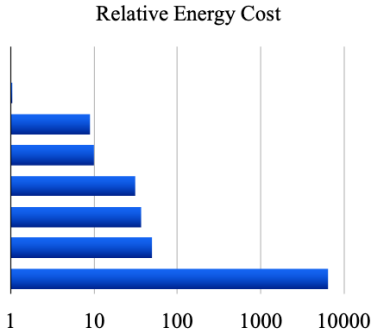
Operational Intensity and the Roofline Model



Data Reuse and Energy

- Each access to main memory consumes > 100x more latency and energy compared to an instruction using only register operands.
- Data reuse important not only for performance…

Operation	Energy [pJ]	Relative Cost
32 bit int ADD	0.1	1
32 bit float ADD	0.9	9
32 bit Register File	1	10
32 bit int MULT	3.1	31
32 bit float MULT	3.7	37
32 bit SRAM Cache	5	50
32 bit DRAM Memory	**640**	**6400**



[Source] Han et al, Learning both Weights and Connections for Efficient Neural Networks

Data Reuse and Batching

- MxM allows a **higher degree of weight reuse** compared to MxV
  - Increased Op. Intensity

- Consequently, when possible, **batching** is one of the easiest (yet quite effective) ways to increase inference **throughput** and **energy efficiency.**

Batching as a Computational Optimization

- Naïve implementation (no batching)

```
for i in range(W.shape()[0]):

    y_i = 0

     for j in range(W.shape()[1]):
          w_ij = w[i,j]
          x_j = x[j]
          y_i = y_i

     y[i] = y_i
```

- Inner loop: 2 loads, 1 MAC.
- **Op. Intensity = 0.125 (assuming 4-byte data)**

Implicit assumption: enough internal memory (CPU registers) to keep all inputs and partial outputs

- Naïve implementation (batch of 4)

```
for i in range(W.shape()[0]):
          y_i0, y_i1, y_i2, y_i3 = 0, 0, 0, 0
     for j in range(W.shape()[1]):
          w_ij = w[i,j]
          x_j0 = x[j,0], x_j1 = x[j,1]
          x_j2 = x[j,2], x_j3 = x[j,3]

          y_i0 = y_i0 + w_ij * x_j0
          y_i1 = y_i1 + w_ij * x_j1
          y_i0 = y_i2 + w_ij * x_j2
          y_i1 = y_i3 + w_ij * x_j3

     y[i,0] = y_i0, y[1,1] = y_i1
     y[i,2] = y_i2, y[1,3] = y_i3
```

- Inner loop: 5 loads, 4 MACs.
- **Op. Intensity = 0.2**

Further Data Reuse

- The concept of “data reuse” can be extended further:
  - To both inputs and weights
  - Also without batching…
- In this example:
  - Unroll over N = 4 inputs
  - 2 weights loaded in each inner iteration
  - 8 MACs with 6 Loads
  - Op. Intensity = 0.33

```
for i in range(0, W.shape()[0], 2):
    y_i0, y_i1, y_i2, y_i3 = 0, 0, 0, 0
    y_ii0, y_ii1, y_ii2, y_ii3 = 0, 0, 0, 0
    for j in range(W.shape()[1]):
        w_ij = w[i,j]
        w_iij = w[i+1,j]
        x_j0 = x[j,0], x_j1 = x[j,1]
        x_j2 = x[j,2], x_j3 = x[j,3]

        # … etc …
```

Fully Connected Layer

- The theoretical OI limit tents to B, and would be obtained if we could keep all data in internal registers:

$$OI \leq \frac{Min.\ number\ of\ ops}{Min.\ number\ of\ loads + stores} = \frac{MN * B}{MN + (M + N) * B}$$

Weight loads

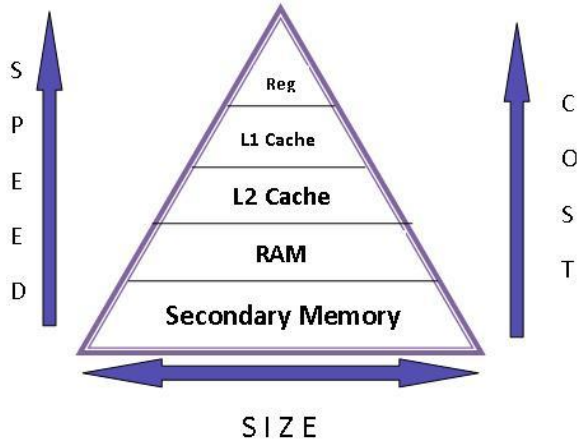
Input loads

Output stores

- Not possible, at least in the innermost memory levels…

Hierarchical Data Reuse

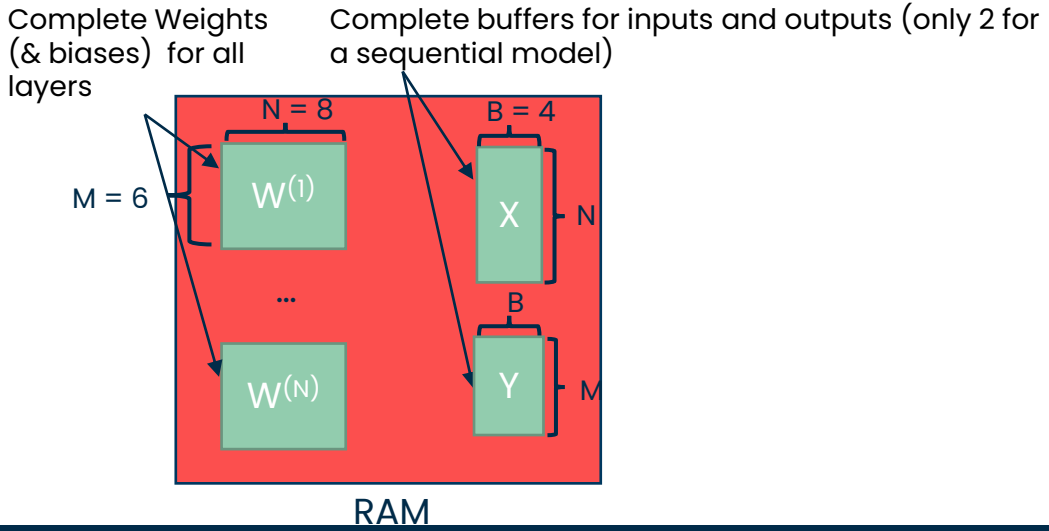
- Modern computers are not just CPU and Main Memory
  - Multiple levels of HW caches or SW-controlled Scratchpad Memories in-between



- The concept of data reuse must be applied **hierarchically**
  - So-called (temporal) **layer tiling**

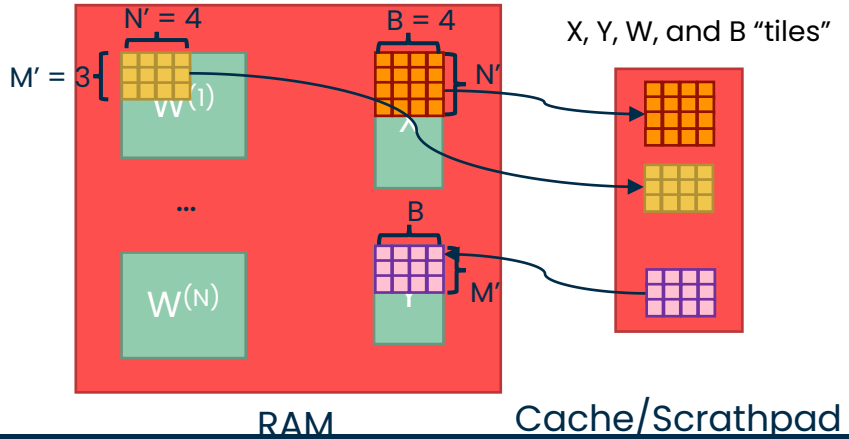
Temporal Tiling Example

- The extent to which you can apply data reuse at each level depends on the size of the corresponding memory
- Conceptual Example (one of the many possible reuse strategies):
  - Consider Layer 1 with M=6 output neurons, N=8 input neurons, and batch size B=4 (neglect bias for simplicity)



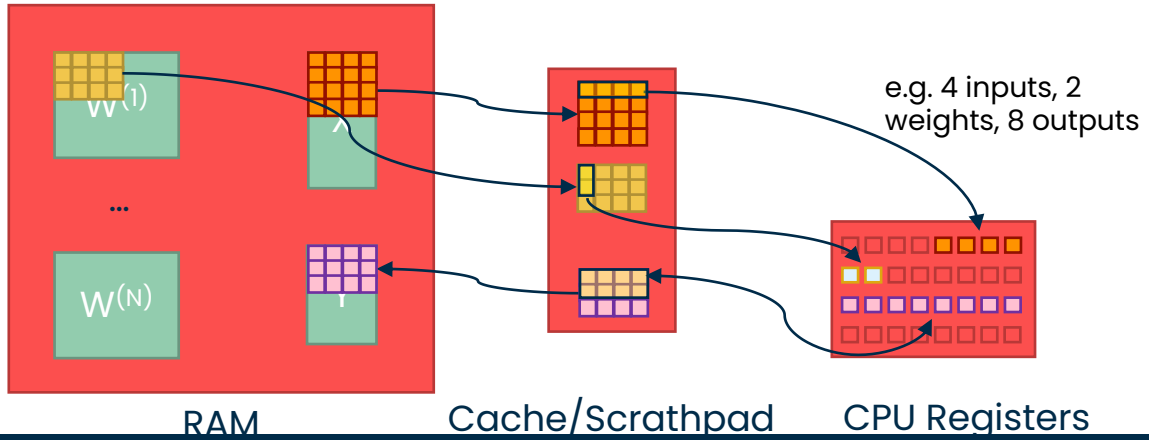
Temporal Tiling Example

- The extent to which you can apply data reuse at each level depends on the size of the corresponding memory
- Conceptual Example (one of the many possible reuse strategies):
  - Consider Layer 1 with M=6 output neurons, N=8 input neurons, and batch size B=4 (neglect bias for simplicity)



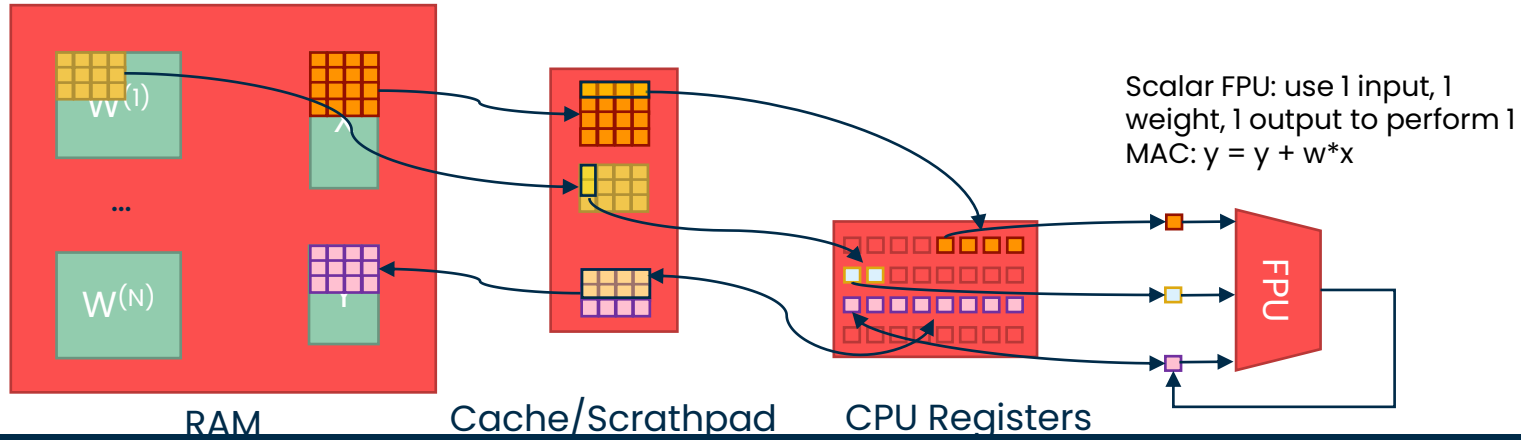
Temporal Tiling Example

- The extent to which you can apply data reuse at each level depends on the size of the corresponding memory
- Conceptual Example (one of the many possible reuse strategies):
  - Consider Layer 1 with M=6 output neurons, N=8 input neurons, and batch size B=4 (neglect bias for simplicity)



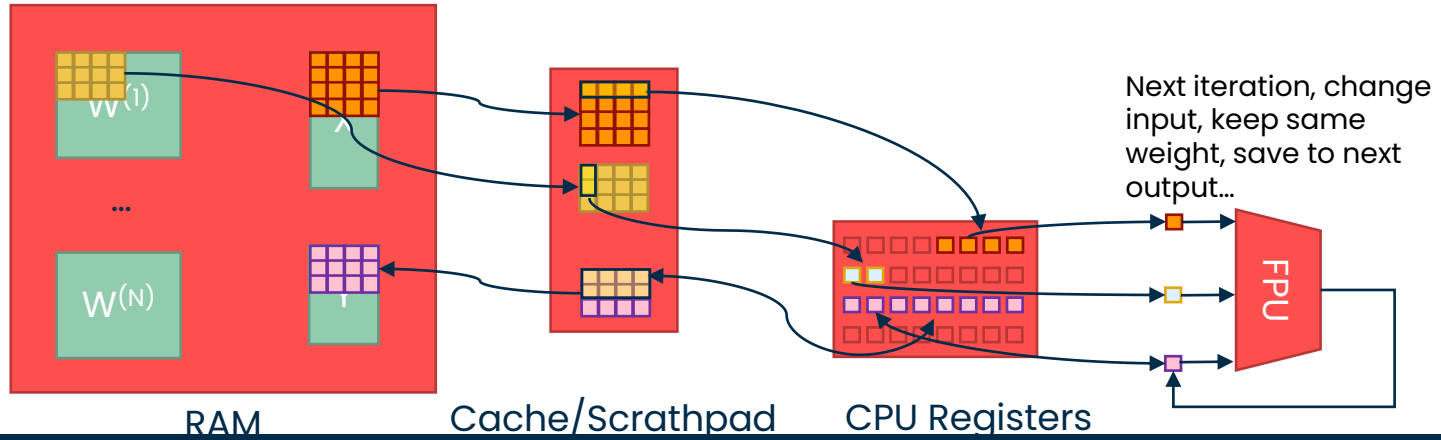
Temporal Tiling Example

- The extent to which you can apply data reuse at each level depends on the size of the corresponding memory
- Conceptual Example (one of the many possible reuse strategies):
  - Consider Layer 1 with M=6 output neurons, N=8 input neurons, and batch size B=4 (neglect bias for simplicity)



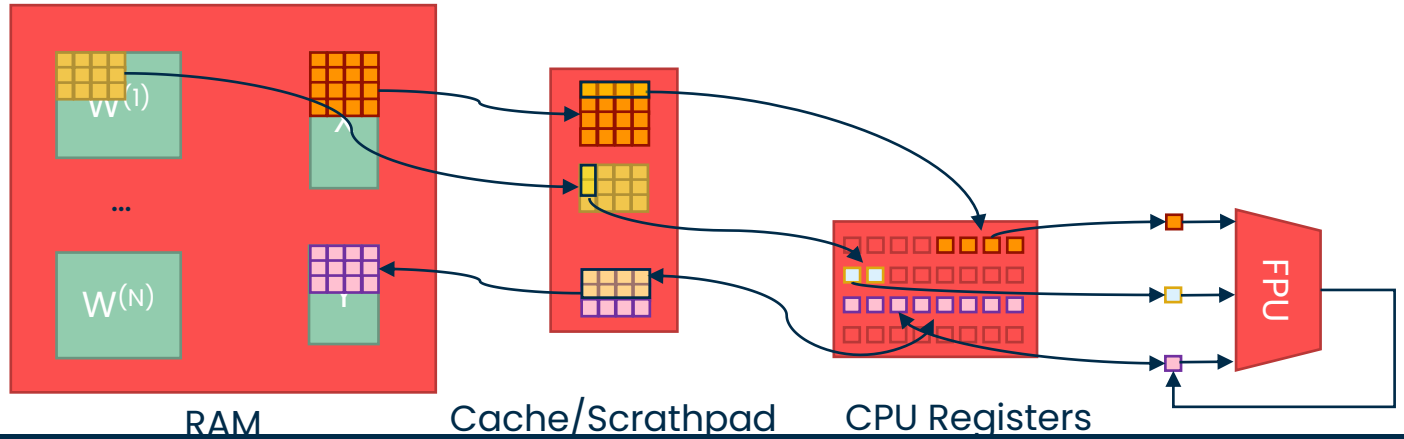
Temporal Tiling Example

- The extent to which you can apply data reuse at each level depends on the size of the corresponding memory
- Conceptual Example (one of the many possible reuse strategies):
  - Consider Layer 1 with M=6 output neurons, N=8 input neurons, and batch size B=4 (neglect bias for simplicity)



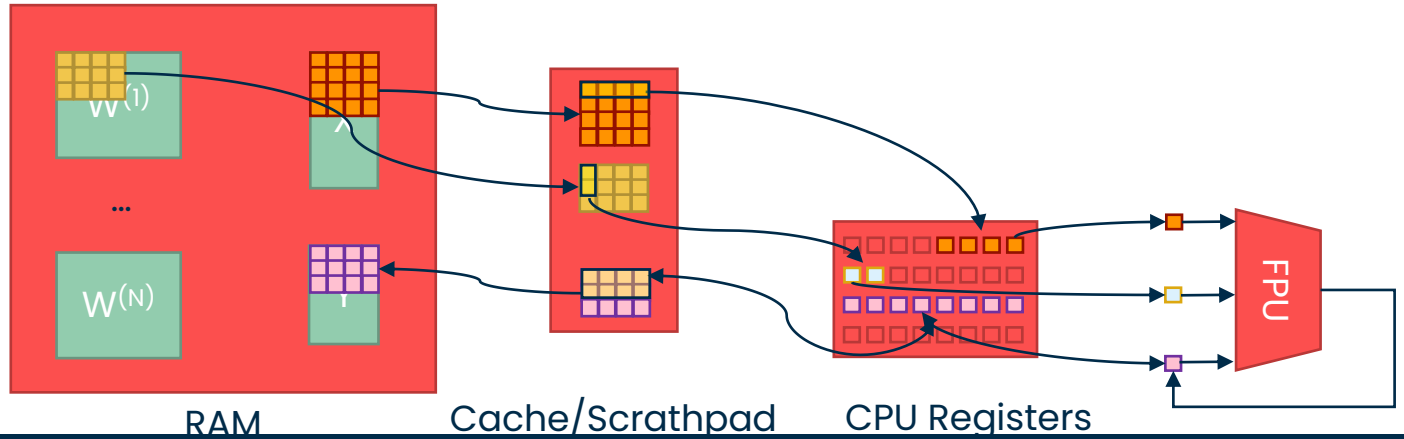
Temporal Tiling Example

- The extent to which you can apply data reuse at each level depends on the size of the corresponding memory
- Conceptual Example (one of the many possible reuse strategies):
  - Consider Layer 1 with M=6 output neurons, N=8 input neurons, and batch size B=4 (neglect bias for simplicity)



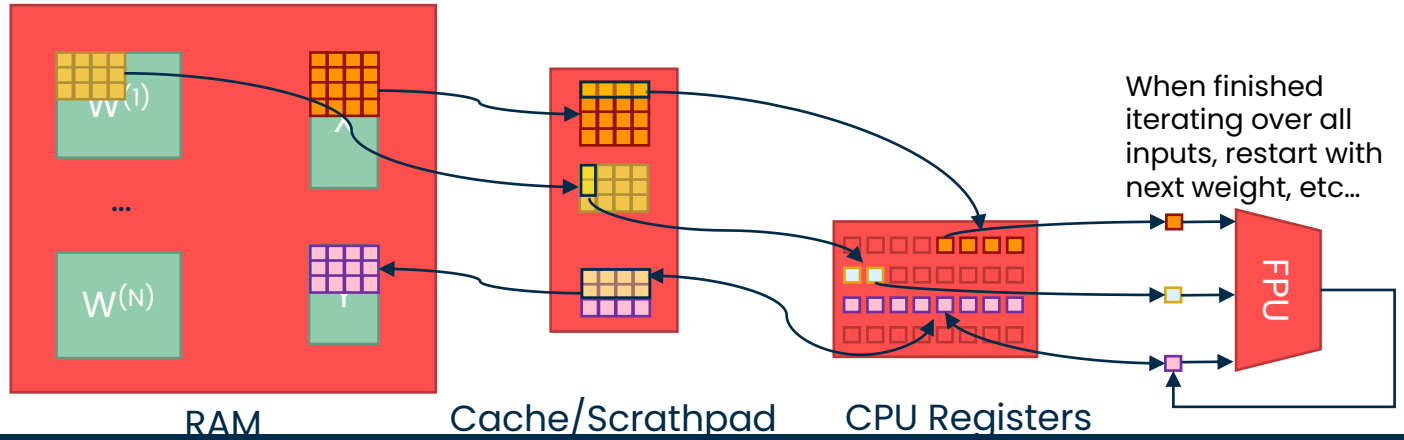
Temporal Tiling Example

- The extent to which you can apply data reuse at each level depends on the size of the corresponding memory
- Conceptual Example (one of the many possible reuse strategies):
  - Consider Layer 1 with M=6 output neurons, N=8 input neurons, and batch size B=4 (neglect bias for simplicity)



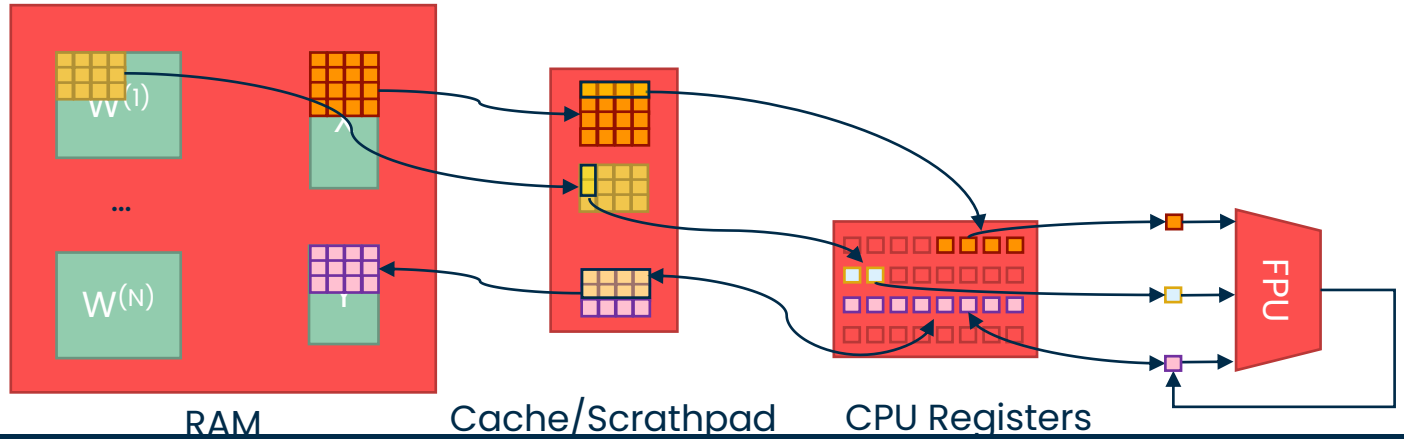
Temporal Tiling Example

- The extent to which you can apply data reuse at each level depends on the size of the corresponding memory
- Conceptual Example (one of the many possible reuse strategies):
  - Consider Layer 1 with M=6 output neurons, N=8 input neurons, and batch size B=4 (neglect bias for simplicity)



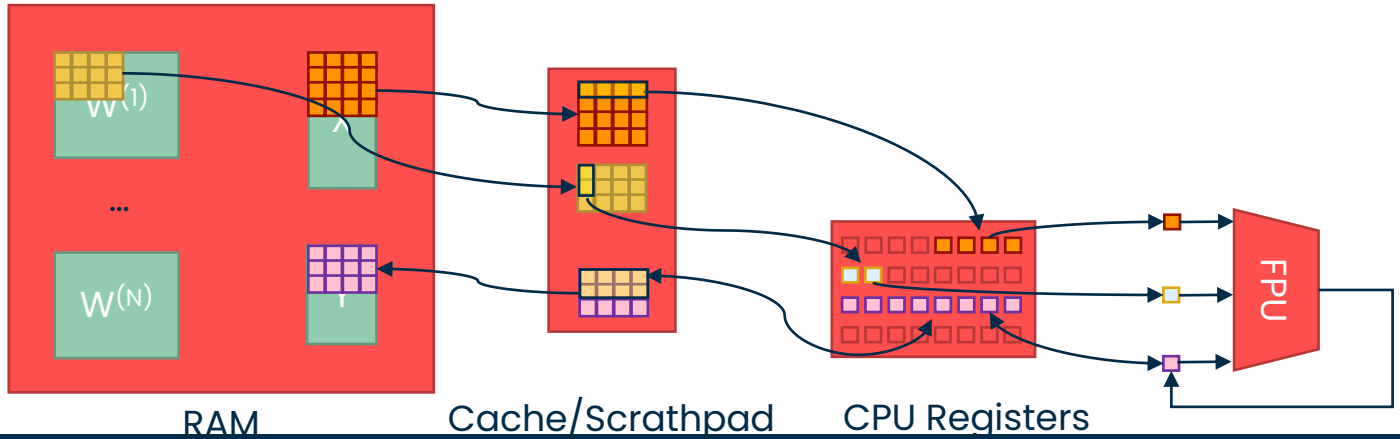
Temporal Tiling Example

- The extent to which you can apply data reuse at each level depends on the size of the corresponding memory
- Conceptual Example (one of the many possible reuse strategies):
  - Consider Layer 1 with M=6 output neurons, N=8 input neurons, and batch size B=4 (neglect bias for simplicity)



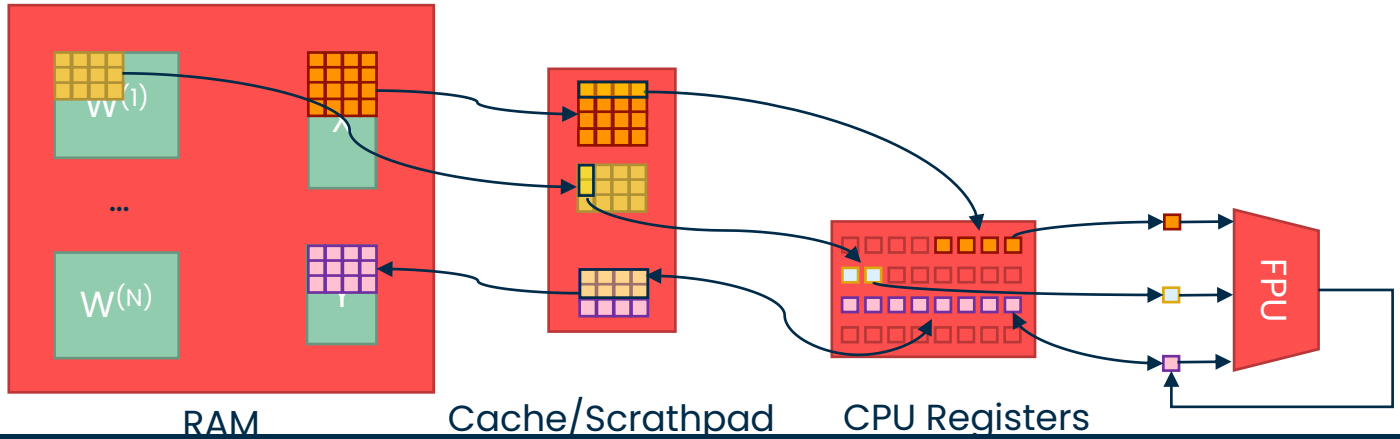
Temporal Tiling Example

- The extent to which you can apply data reuse at each level depends on the size of the corresponding memory
- Conceptual Example (one of the many possible reuse strategies):
  - Consider Layer 1 with M=6 output neurons, N=8 input neurons, and batch size B=4 (neglect bias for simplicity)



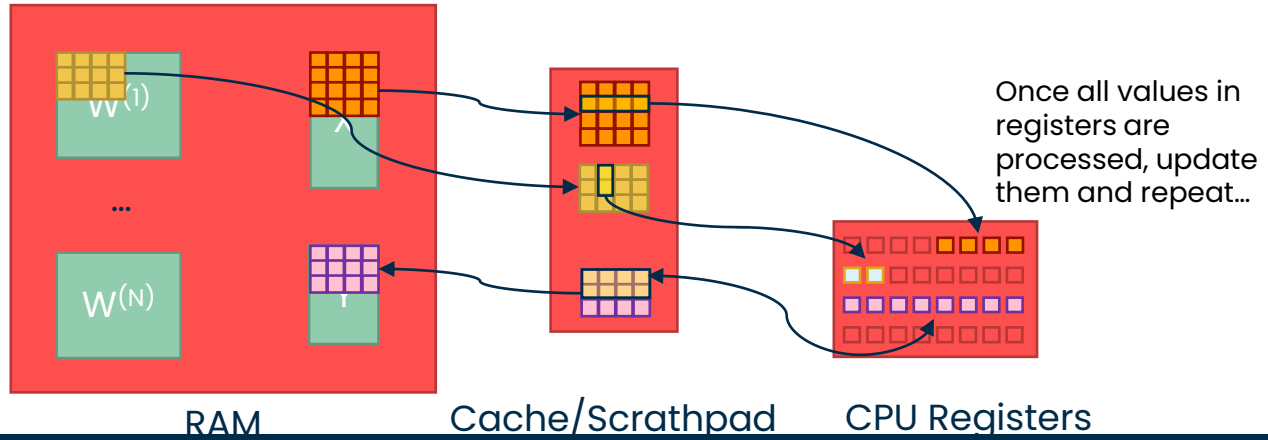
Temporal Tiling Example

- The extent to which you can apply data reuse at each level depends on the size of the corresponding memory
- Conceptual Example (one of the many possible reuse strategies):
  - Consider Layer 1 with M=6 output neurons, N=8 input neurons, and batch size B=4 (neglect bias for simplicity)



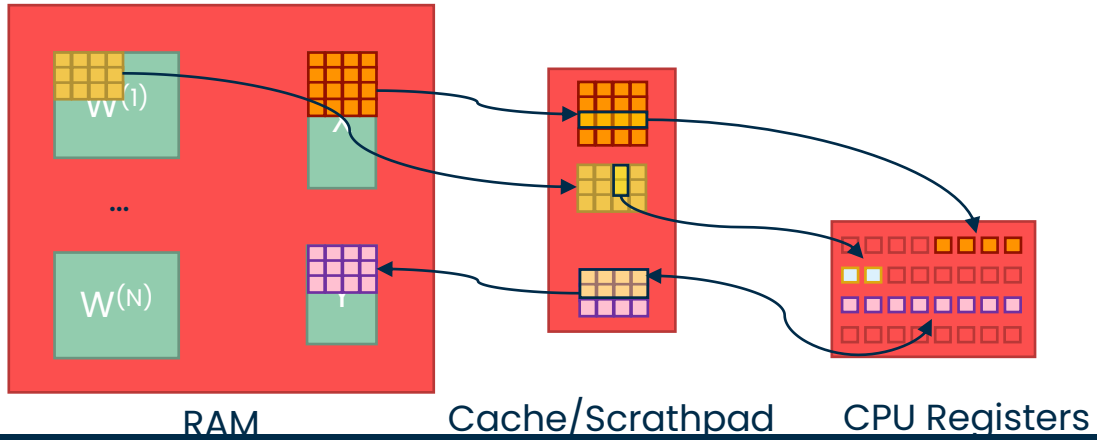
Temporal Tiling Example

- The extent to which you can apply data reuse at each level depends on the size of the corresponding memory
- Conceptual Example (one of the many possible reuse strategies):
  - Consider Layer 1 with M=6 output neurons, N=8 input neurons, and batch size B=4 (neglect bias for simplicity)



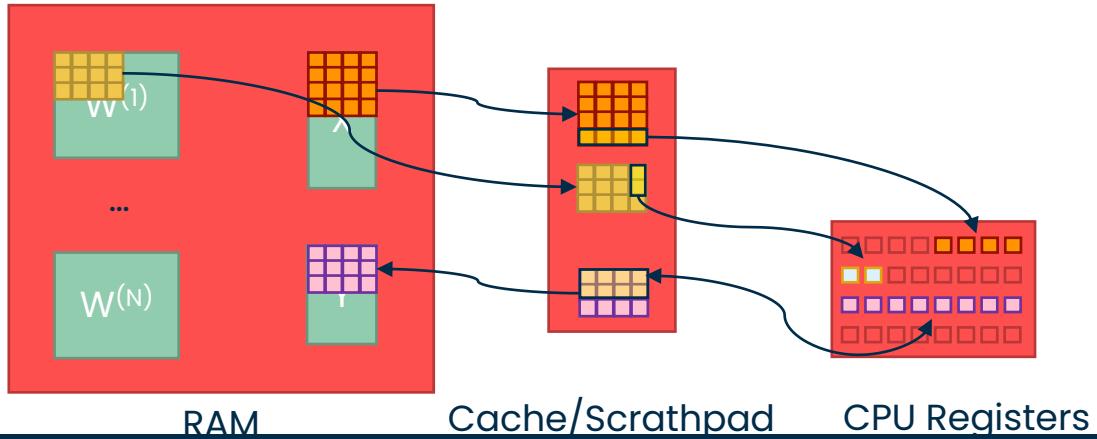
Temporal Tiling Example

- The extent to which you can apply data reuse at each level depends on the size of the corresponding memory
- Conceptual Example (one of the many possible reuse strategies):
  - Consider Layer 1 with M=6 output neurons, N=8 input neurons, and batch size B=4 (neglect bias for simplicity)



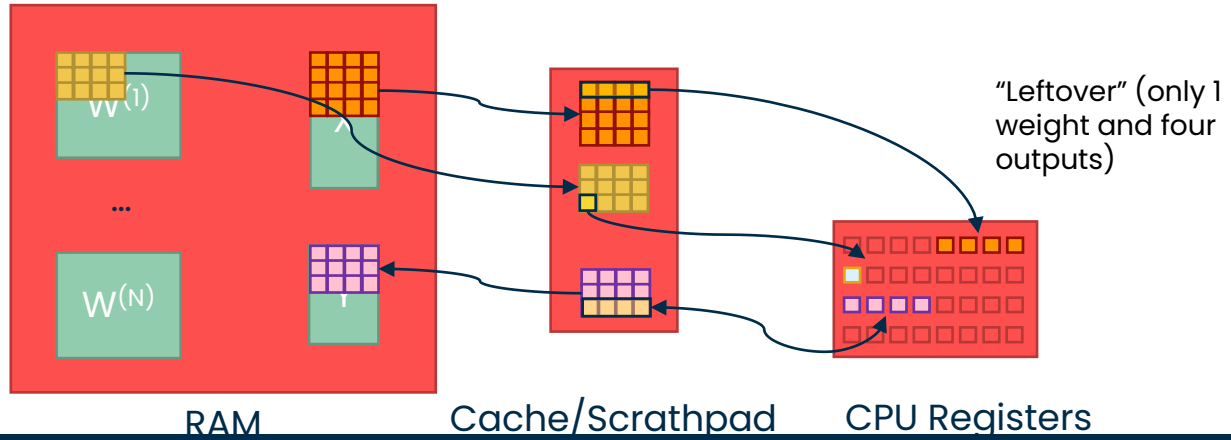
Temporal Tiling Example

- The extent to which you can apply data reuse at each level depends on the size of the corresponding memory
- Conceptual Example (one of the many possible reuse strategies):
  - Consider Layer 1 with M=6 output neurons, N=8 input neurons, and batch size B=4 (neglect bias for simplicity)



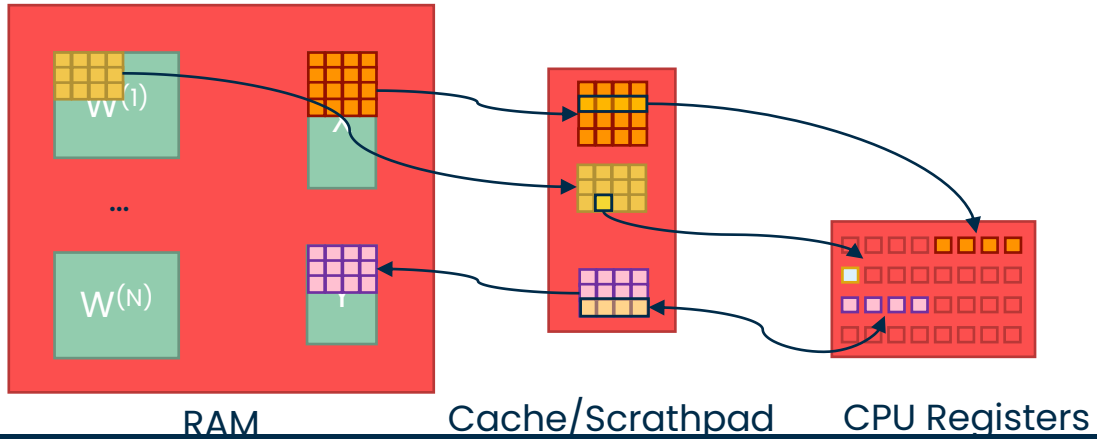
Temporal Tiling Example

- The extent to which you can apply data reuse at each level depends on the size of the corresponding memory
- Conceptual Example (one of the many possible reuse strategies):
  - Consider Layer 1 with M=6 output neurons, N=8 input neurons, and batch size B=4 (neglect bias for simplicity)



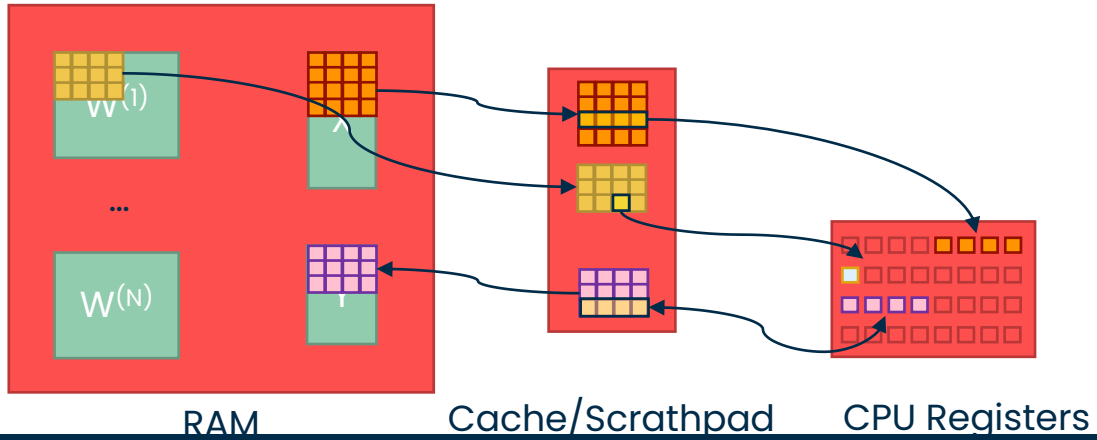
Temporal Tiling Example

- The extent to which you can apply data reuse at each level depends on the size of the corresponding memory
- Conceptual Example (one of the many possible reuse strategies):
  - Consider Layer 1 with M=6 output neurons, N=8 input neurons, and batch size B=4 (neglect bias for simplicity)



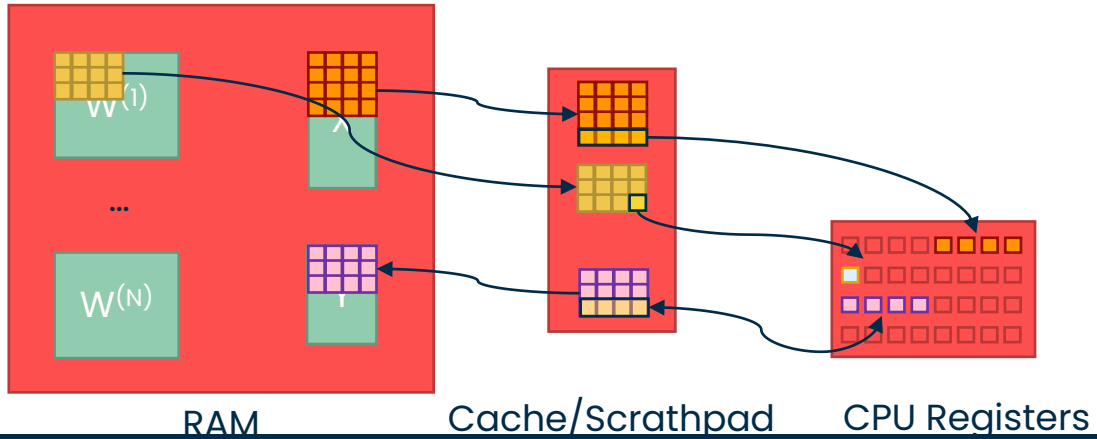
Temporal Tiling Example

- The extent to which you can apply data reuse at each level depends on the size of the corresponding memory
- Conceptual Example (one of the many possible reuse strategies):
  - Consider Layer 1 with M=6 output neurons, N=8 input neurons, and batch size B=4 (neglect bias for simplicity)



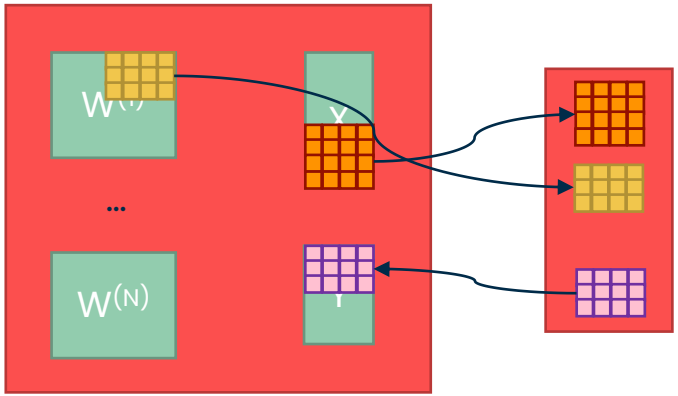
Temporal Tiling Example

- The extent to which you can apply data reuse at each level depends on the size of the corresponding memory
- Conceptual Example (one of the many possible reuse strategies):
  - Consider Layer 1 with M=6 output neurons, N=8 input neurons, and batch size B=4 (neglect bias for simplicity)



Temporal Tiling Example

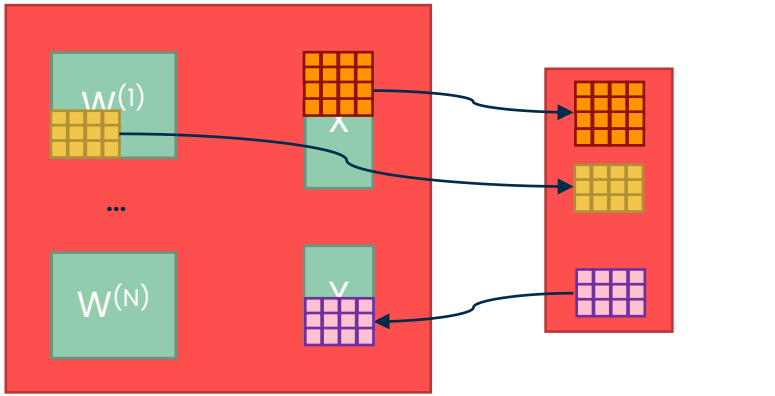
- The extent to which you can apply data reuse at each level depends on the size of the corresponding memory
- Conceptual Example (one of the many possible reuse strategies):
  - Consider Layer 1 with M=6 output neurons, N=8 input neurons, and batch size B=4 (neglect bias for simplicity)



Once all values in L1 memory are processed, change tiles in L2 and repeat...

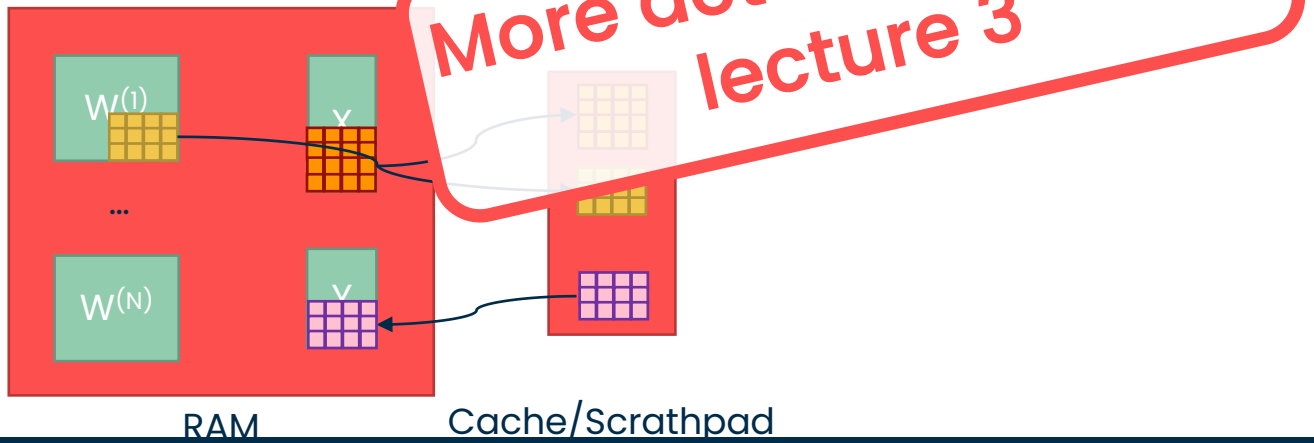
Temporal Tiling Example

- The extent to which you can apply data reuse at each level depends on the size of the corresponding memory
- Conceptual Example (one of the many possible reuse strategies):
  - Consider Layer 1 with M=6 output neurons, N=8 input neurons, and batch size B=4 (neglect bias for simplicity)



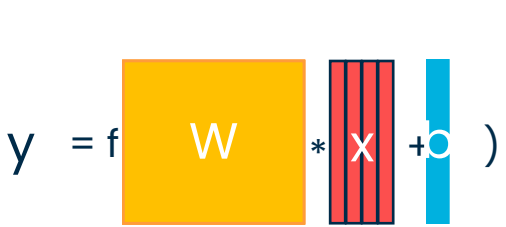
Temporal Tiling Example

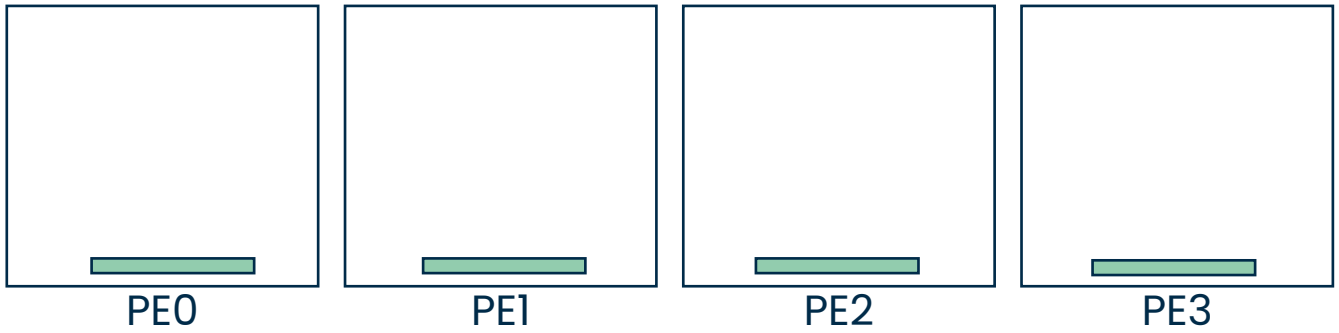
- The extent to which you can apply data reuse at each level depends on the size of the corresponding memory
- Conceptual Example (one of the many possible reuse strategies):
  - Consider Layer 1 with M=6 output neurons, N=8 input neurons, and batch size B=4 (neglect bias for simplicity)



Parallelizing MatMul

- Split computation over multiple Processing Elements (PEs).
  - E.g. cluster cores in GAP8/9
  - Also called "**spatial tiling**".
- Can be done over multiple axes.
  - Example: **data parallelism** → Split over N dimension, replicate (broadcast) **W** and **b** tensors

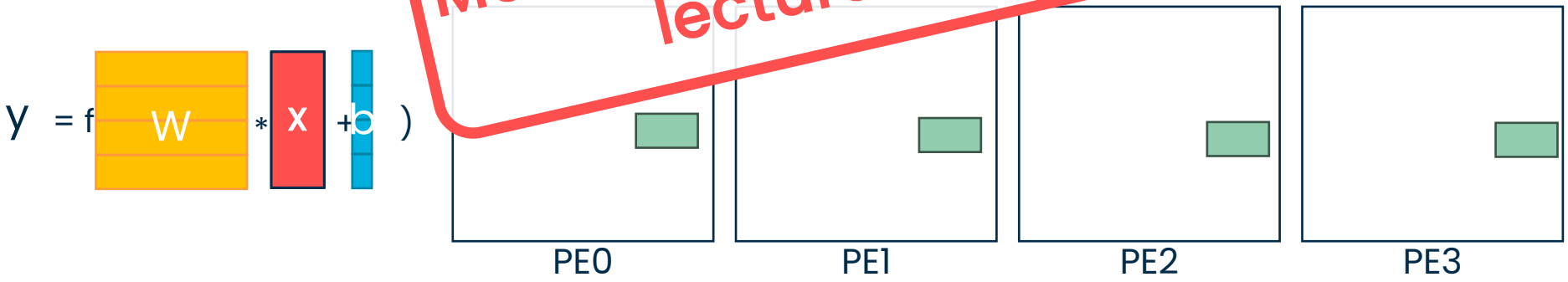




Parallelizing MatMul

- Split computation over multiple Processing Elements (PEs).
  - E.g. cluster cores in GAP8/9
  - Also called “spatial tiling”.
- Can be done over multiple axes.
  - Example: **model parallelism** over **W** and **b** rows, split over K dimension, broadcast **x** tensor

More details on this in lecture 3



Batching and Inference (Reprise)

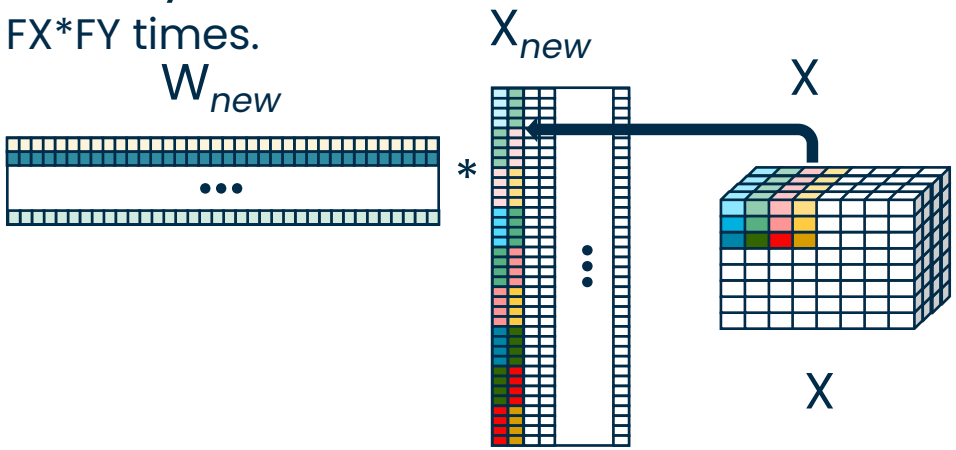
- As we've seen, processing a batch of inputs is beneficial for efficiency
  - More data reuse opportunities
  - Enables data-parallel computation.

- But batching increases **throughput** (and energy efficiency) at the expense of **latency**.
  - Not possible in **latency-sensitive** applications.

- Many edge applications are latency-sensitive
  - Examples: autonomous driving, real-time audio processing, etc.

Conv as MatMul (Reprise)… Why?

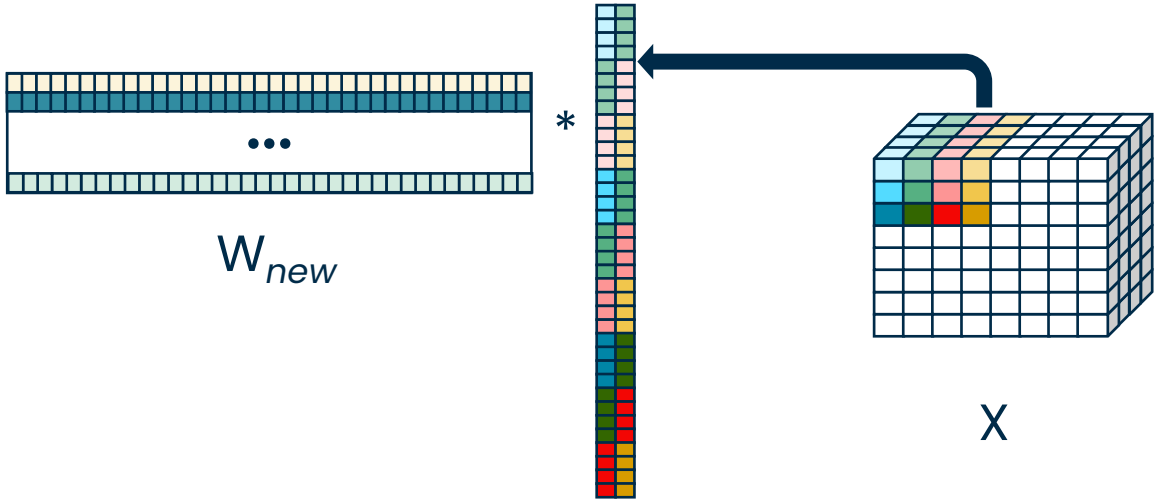
- The im2col process causes a memory overhead:
  - Each input “pixel” is replicated up to FX*FY times.



- But im2col + MatMul has efficiency advantages:
  - Leverage optimized linear algebra routines for GEMM (BLAS/cuBLAS)
  - More regular **memory access pattern**:
    - Easier to use SIMD instructions (both arithmetic and load/store)
    - …or specialized instructions like hardware loops and autoincrement

Extra: Conv as MatMul… Panel Dot Product

- Libraries for constrained devices (e.g. CMSIS-NN, PULP-NN) often implement a "partial" im2col (Panel Dot Product):
  - Materialize only a few rows of the im2col matrix (e.g., 1, 2 or 4)
  - Balance **efficiency gain** and **input activations reuse** opportunity with **memory overhead** and **registers** occupation.



Tensor Layouts in Memory

- An important choice when implementing low-level software for DNNs is the data layout in memory.

Horizontally adjacent pixels of the **same channel** are contiguous in memory

- For instance, input/output activations in 2[illegible]Ns are typically memorized with one of two layouts:
  - **(N)CHW** or Channels-first → `x[C][OY][OX]`
  - **(N)HWC** or Channels-last → `x[OY][OX][C]`

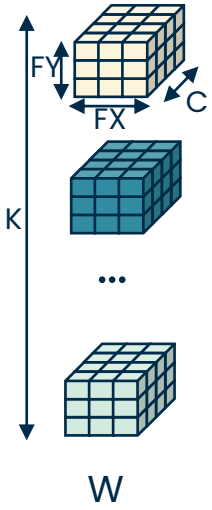
Consecutive channels for the **same pixel** are continuous in memory

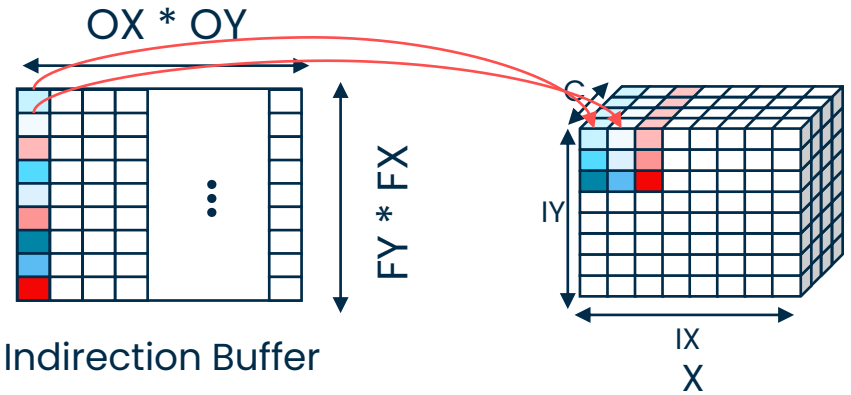
  - Both have advantages and disadvantages, but HWC enables faster im2col memcpy with SIMD loads, and is the one used in our target library

- Weights, in our case, use a CoHWCi layout: `W[K][FY][FX][C]`

Alternative Conv Implementation

- **Indirect Convolution** (e.g., QNNPACK)
  - Store a **pointer** to the start of each (H, W) location
  - Reduce memory overhead by a factor C_{in} (no "im2col")
  - At the cost of a "less regular" MatMul loop





Conv Implementations

- Pseudo-code for a single input location:
  - Assuming a 1 x 1 (1 input location, 1 output channel) innermost kernel:

Im2col + PDOT

```
//Im2col Loop
for ix in [x-FX/2, x+FX/2]:
  for iy in [x-FY/2,  x+FY/2]:
    for ci in C:
      x_i2c[i] = x[(ix + OX*iy)*C + ci]
      i++
//MatMul loop
for co in [0: K]:
  for i in [0:FX*FY*C]:
    out[co, i] += w[co, i] * x_i2c[i]
```

Indirect Conv

```
//Indirect Buffer Loop
for ix in [x-FX/2, x+FX/2]:
  for iy in [y-FY/2,  y+FY/2]:
    x_ind[i] = x + (ix + OX*iy)*C
    i++
//MatMul loop
for co in [0: K]:
  for i in [0:FX*FY]:
    for ci in [0:C]:
      out[co, i*C + ci] += w[co, i] *
                           x[[x_ind[i] + ci]
```

Conv Implementations

- Pseudo-code for a single input location:
  - Assuming a 1 x 1 (1 input location, 1 output channel) innermost kernel:

Im2col + PDOT

```
//Im2col Loop
for ix in [x-FX/2, x+FX/2]:
  for iy in [x-FY/2,  x+FY/2]:
```

Total cycles (im2col + PDOT):

FX*FY*C*
[I2C_1stLoop + **K** * I2C_2ndLoop]

```
//MatMul loop
for co in [0: K]:
  for i in [0:FX*FY*C]:
    out[co, i] += w[co, i] * x_i2c[i]
```

Indirect Conv

```
//Indirect Buffer Loop
for ix in [x-FX/2, x+FX/2]:
  for iy in [y-FY/2,  y+FY/2]:
```

Total cycles (Indirect):

FX*FY*C*
[(1/C)*Ind_1stLoop + **K** * Ind_2ndLoop]

```
for co in [0: K]:
  for i in [0:FX*FY]:
    for ci in [0:C]:
      out[co, i*C + ci] += w[co, i] *
                           x[[x_ind[i] + ci]
```

Conv Implementations

- For large Cout (e.g., 32), the im2col loop becomes negligible.
  - **Im2col + PDOT** ≈ FX*FY*C* K * **I2C_2ndLoop**
  - **Indirect** ≈ FX*FY*C* K * **Ind_2ndLoop**

What matters are the the MatMul loops

- The additional inner loop overheads (initialize, compare&branch, etc) and memory access often make Indirect Conv. slower

- Difference may depend on HW support
  - Load instructions with post-increment, HW loops, etc.

```
for co in [0: K]:
  for i in [0:FX*FY*C]:
    out[co, i] += w[co, i] * x_i2c[i]
```

VS

```
for co in [0: K]:
  for i in [0:FX*FY]:
    for ci in [0:C]:
      out[co, i*C + ci] += w[co, i]
                           *x[[x_ind[i] + ci]
```

Summary

- **Executing DNN Layers efficiently.**
  - Maximize **data reuse** and **parallelism**
  - Exploit the characteristics of the HW platform
    - Memory hierarchy, available HW support (SIMD, multicore, etc).

- Goes beyond MatMul….
  - E.g. for Transformers, certain non-MatMul operations can be critical
  - Softmax, LayerNorm, etc

- Lecture 3 will focus on this more in detail…

Politecnico di Torino

Strategy 2

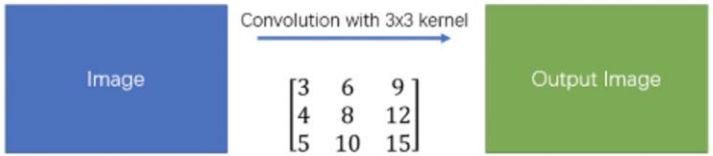
A First Example: Efficient DNN Layers

Efficient Layers

- One effective way to optimize DNNs is to replace standard layers with more efficient alternatives
  - Approximating the original layer's output
- Well-studied for Convolutions historically, and recently for Transformers (MHSA)
- We'll see just a couple of examples here…

Spatially Separable Convolutions

- **Spatially Separable Convolutions** (earliest embodiment):
  - Reduce a KxK 2D Conv to the combination of Kx1 and 1xK 1D Convs
  - E.g., for K=3, reduce the number of MULs from **3*3*C_{in}** to **2*(3*1)*C_{in}**



[Source] Wang, A Basic Introduction to Separable Convolutions.

Depthwise Separable Convolutions

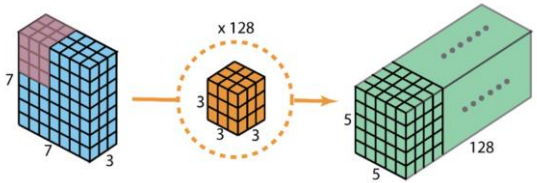
- Spatially Separable Conv. are not very common:
  - Very few KxK kernels are spatially separable onto a combination of Kx1 and 1xK
  - Training on Kx1 + 1xK directly yields poor accuracy.

- **Depthwise Separable Conv** are much more popular
  - Concept: separate over the depth (i.e., channels) dimension
  - Use an additional **pointwise** (i.e., 1x1) Conv. to combine different channels
  - First made popular by **MobileNets**

Depthwise Separable Convolutions

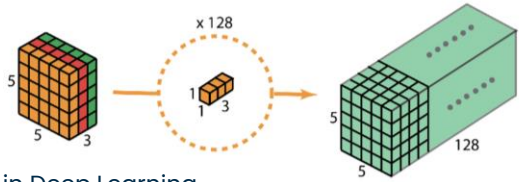
- Separate a multi-channel Conv. into C single-channel convolutions
- Alternate with a pointwise Conv. (i.e. FX=FY=1) to combine info from different channels
- Reduce complexity, often with limited impact on accuracy

Standard Convolution



Depthwise Convolution

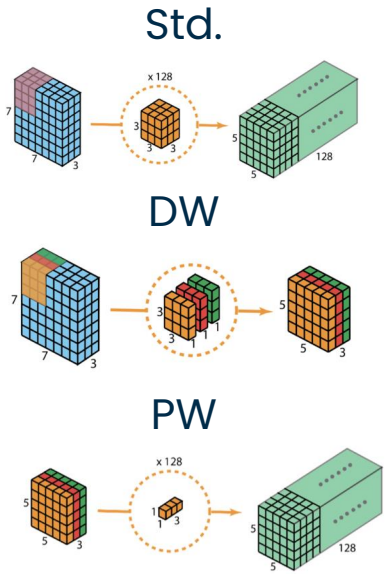
Pointwise (1x1) Convolution



[Source] Bai, K. A Comprehensive Introduction to Different Types of Convolutions in Deep Learning.

Depthwise Separable Convolutions

- Complexity (FLOPs per output location and memory):
  - **Standard Conv:** $O(FX * FY * C * K)$
  - **Depthwise (DW):** $O(FX * FY * C)$
  - **Pointwise (PW):** $O(C * K)$
  - **(DW +PW) / Standard:** $\boldsymbol{O}(\frac{\mathbf{1}}{\boldsymbol{K}} + \frac{\mathbf{1}}{\boldsymbol{FX*FY}})$

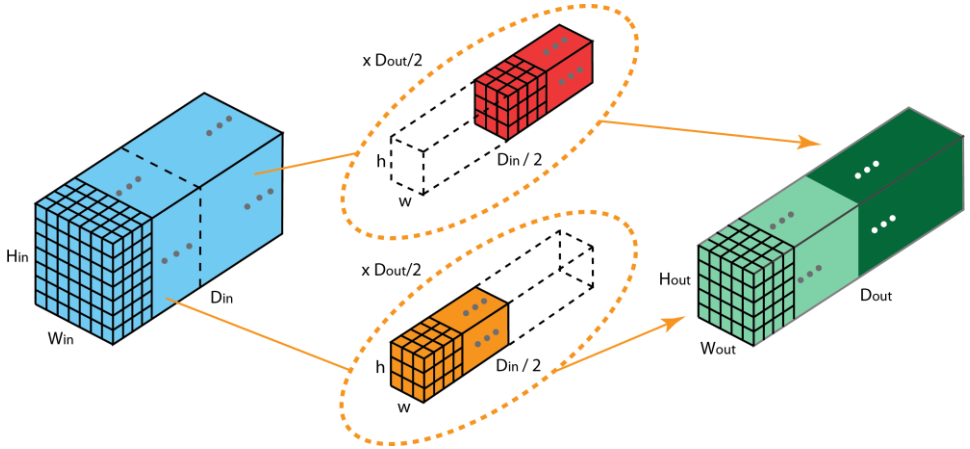


- Example: FX=FY=3, C = 32, K=64
  - **Standard:** $3 * 3 * 32 * 64 = 18432$
  - **DW + PW:** $3 * 3 * 32 + 1 * 1 * 32 * 64 = 288 + 2048 = 2336$ (7.9x reduction!)

- Multiple DW + PW pairs can be shown to approximate any convolution:
  - Guo et al, "Network Decoupling: From Regular to Depthwise Separable Convolutions", 2018

Grouped Convolutions

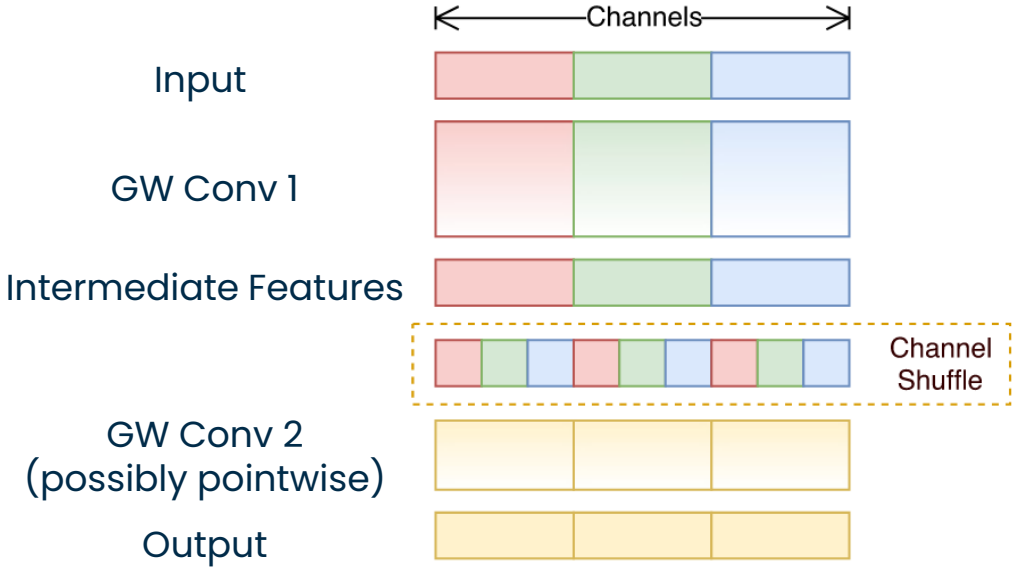
- Grouped or GroupWise (GW) Convolutions are a generalization of DW where each convolution processes C/G input channels
  - Made popular by: X. Zhang et al, *ShuffleNet: An Extremely Efficient Convolutional Neural Network for Mobile Devices*, CVPR 2018
- Complexity: $O(G * (FX * FY * \frac{C}{G} * \frac{K}{G})) = \frac{1}{G} *$ Standard Conv
- Example with G = 2 groups



[Source] Bai, K. A Comprehensive Introduction to Different Types of Convolutions in Deep Learning.

ShuffleNet

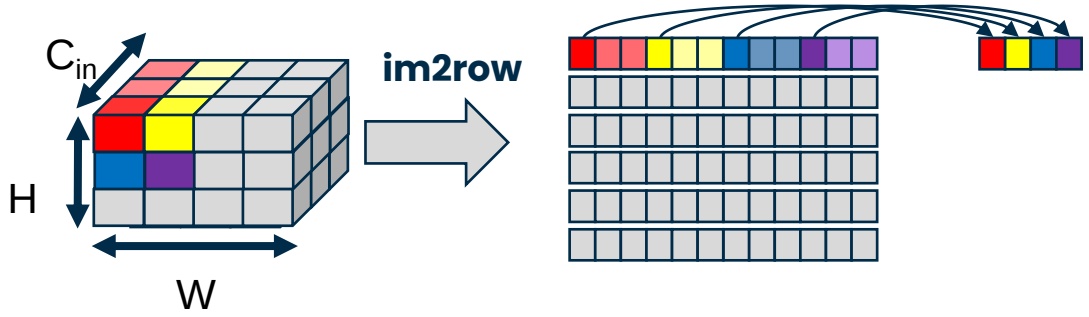
- Grouped Convolutions combined with channel reshuffling so that finally, each output takes information from all input channels:
  - X. Zhang et al, *ShuffleNet: An Extremely Efficient Convolutional Neural Network for Mobile Devices*, 2017



DW/GW Convolutions Efficiency

- DW and GW convolutions clash with the HWC layout typically used for CNNs on CPUs
  - Im2col/Im2row process becomes much slower
- They also have less data reuse opportunities w.r.t. standard conv
- In practice, the theoretical advantages of these layers may reduce significantly!
  - E.g. N times fewer operations << N times spedup….

"On-the-fly" HWC to CHW
Large overhead!



Winograd Convolutions (Extra)

- The complexity of Convolutions can also be simplified by **exact transformations** (not approximated, at least in principle):
  - **FFT:** transform convolution into a multiplication. Only effective for large kernels.
  - **Winograd method:** a generalization of FFT more effective on small kernels
    - [Source] Lavin et al, "Fast Algorithms for Convolutional Neural Networks", 2016

- Example: 1D convolution producing 2 outputs with a FX=3 filter
  - Basic Conv: 6 MUL + 4 ADDs
  - Winograd: 4 MUL + 8 ADDs (g_i combinations can be pre-computed)

$$F(2,3) = \begin{bmatrix} d_0 & d_1 & d_2 \\ d_1 & d_2 & d_3 \end{bmatrix} \begin{bmatrix} g_0 \\ g_1 \\ g_2 \end{bmatrix} = \begin{bmatrix} m_1 + m_2 + m_3 \\ m_2 - m_3 - m_4 \end{bmatrix}$$

where

$$m_1 = (d_0 - d_2)g_0 \qquad m_2 = (d_1 + d_2)\frac{g_0 + g_1 + g_2}{2}$$
$$m_4 = (d_1 - d_3)g_2 \qquad m_3 = (d_2 - d_1)\frac{g_0 - g_1 + g_2}{2}$$

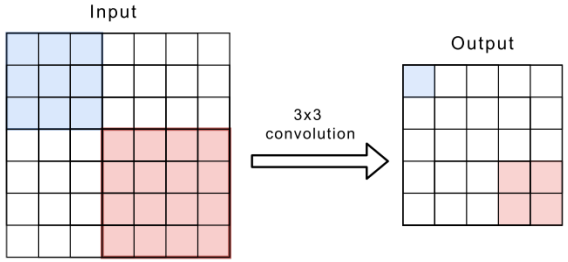
Winograd Convolutions (Extra)

- Can be extended to a multi-channel 2D Conv and any input/kernel size as:

$$f = Z^T(\sum_{c=0}^{C}[(WwW^T)_c \odot (X^T xX)_c])Z$$

  - Where Z, W and X are constant transformation matrices for outputs, weights and inputs, and $\odot$ is the element-wise (Hadamard) product. C are the input channels

- Winograd can work on arbitrary sized input patches, larger than (FX, FY)
  - To produce an output patch of size (n,n), the input is a patch of size (n+FX-1, n+FY-1)



Winograd Convolutions (Extra)

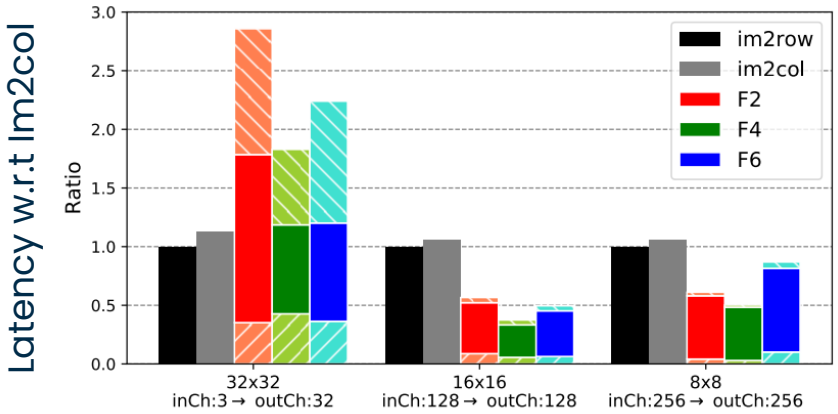
- Winograd reduces the tot. # of MUL in exchange for a higher # of ADD
  - Overall complexity reduction, especially for large (n, n) output patches

$$f = Z^T(\sum_{c=0}^{C}[(WwW^T)_c \odot (X^T xX)_c])Z$$

- Overheads:
  - **Compute:** exctracting $X^T xX$ and $Z^T(res)Z$
  - **Memory:** WwW^T is of size (n+FX-1, n+FY-1), larger than (K, K)
  - **Numerical error:** Winograd suffers from numerical approximation errors. Does not work well with Quantization.
    - On float32, Standard → Winograd conversion has no accuracy impact
    - No longer true on Int-8

Winograd Convolutions (Extra)

- There exist practical implementations of Winograd on CPUs (Cortex-A)
  - Using HWC layout.
- Made compatible with quantization by doing a “Winograd-aware training”
  - Using Winograd transformations already at training time.
  - Allowing matrices W, X and Z to be learned
- Speed-ups lower than theory, but still present (up to 1.3x on A73):



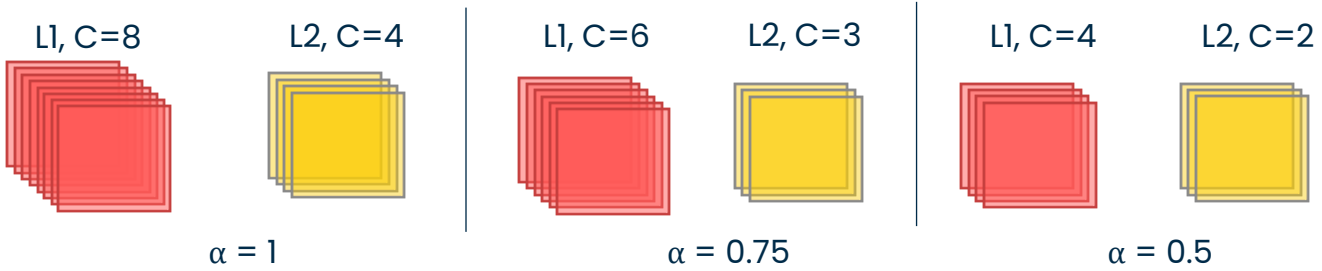
2x2, 4x4 and 6x6 Winograd output patches

[Source] Ferndandes-Marques et al, Searching for Winograd-aware Quantized Networks, 2020

Width and Resolution Multipliers

- Besides DW + PW Conv, MobileNetV1 also included two other manual ways to reduce complexity (at the expense of accuracy):

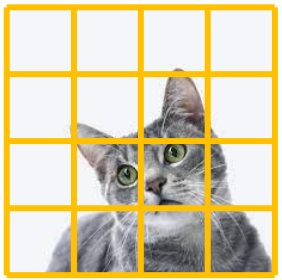
- **Width multiplier (α):** uniform reduction of the # of channels in all layers
  - Sort of "manual channel-based pruning"
  - Reduce DW Conv params and ops by α and PW conv by α^2
  - Reduce params and ops of final FC layers



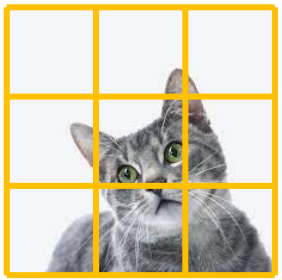
Width and Resolution Multipliers

- Besides DW + PW Conv, MobileNetV1 also included two other manual ways to reduce complexity (at the expense of accuracy):

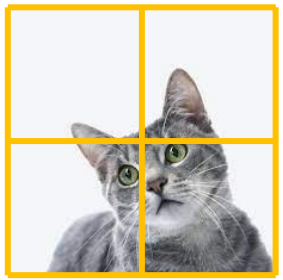
- **Resolution multiplier (ρ)**: input resolution down-scaling
  - Reduce the n. ops of all subsequent Conv layers by ρ^2
  - Reduce the params and ops of final FC layers



$\rho = 1$



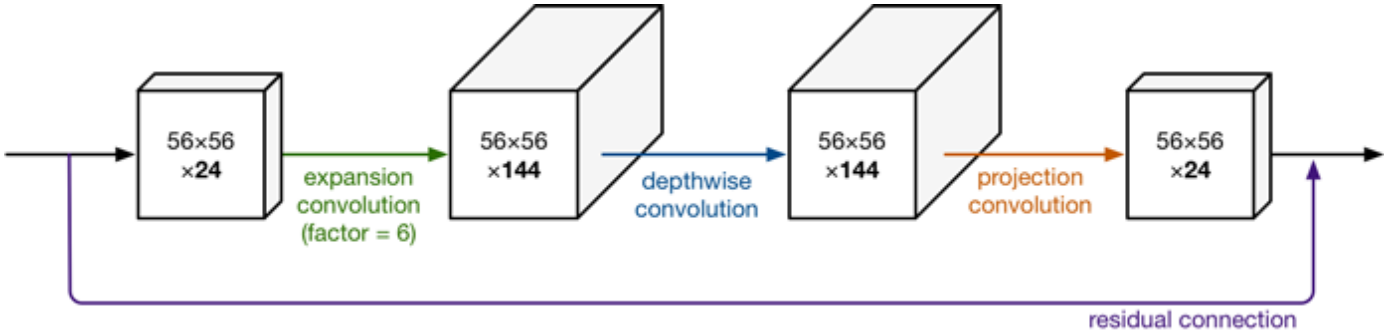
$\rho = 0.75$



$\rho = 0.5$

MobileNet V2 and later

- MobileNets V2 added further (manual) optimizations to the DW + PW structure of V1:
  - Expansion and projection layers
  - Residual connections



Other Efficient Layers

- Attention:
  - **Low-rank Transformer, Linformer, Performer, etc:** reduce complexity vs sequence length to sub-quadratic
    - Y. Tay et al, “Efficient Transformers: A Survey”, arXiv

- Out of scope…

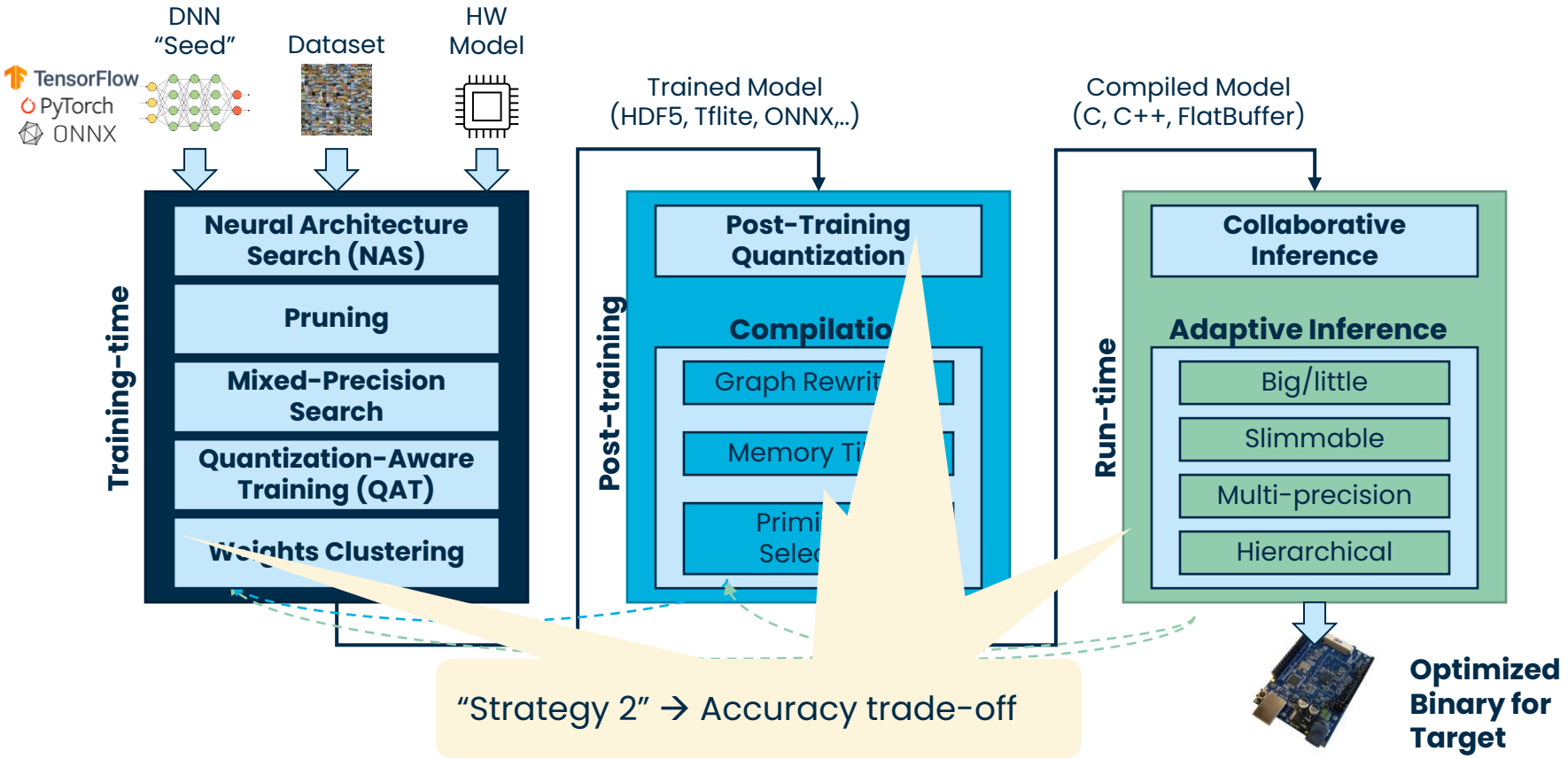
Other Efficient Layers

- **Key Question:** efficient layers such as DW, GW & co, width/res. Multipliers, etc, all **can affect accuracy**.
- When to use them (and when not)?
- **From hand-tuned optimization to design automation…**

Politecnico
di Torino

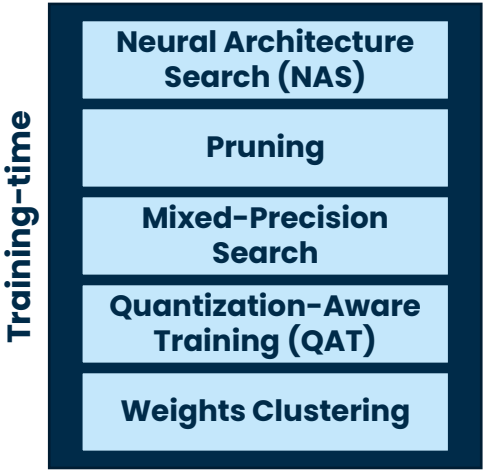
An Automated DNN Optimization Flow

Automated DNN Optimization and Deployment Flow



Automated DNN Optimization and Deployment Flow

- **NAS:**
  - Automatically select architecture & hyperparmeters
  - Co-optimize task accuracy and complexity, i.e., memory, latency, energy
- **Pruning:**
  - Eliminate unimportant portions of the network
- **Mixed-Precision Search and QAT:**
  - Find the best data precision for each computation
  - Compensate the accuracy drop due to quantization during training
- **Weights Clustering:**
  - Reduce the number of distinct weight values



TensorFlow Lite PyTorch Vizier

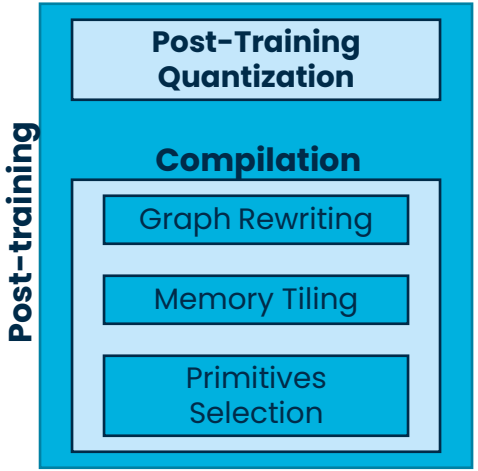
Neural Network Intelligence

PLiNIO
Plug-and-play Lightweight Neural Inference Optimization

Automated DNN Optimization and Deployment Flow

- **Post-Training Quantization, Pruning, etc:**
  - Alternative to training-time optimizations, e.g. when training is not feasible

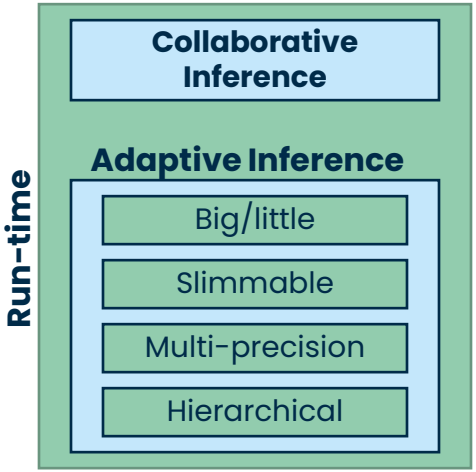
- **DNN Compilation:**
  - Translate high-level model (e.g. ONNX) to optimized C
  - Exploit properties of DNN kernels in compiler passes:
    - E.g., Operator fusion, temporal/spatial tiling, etc.
  - Exploit available accelerators
  - Etc.



Automated DNN Optimization and Deployment Flow

- **Collaborative Inference:**
  - Share (part of) the execution with higher-level devices
    - Edge servers or cloud

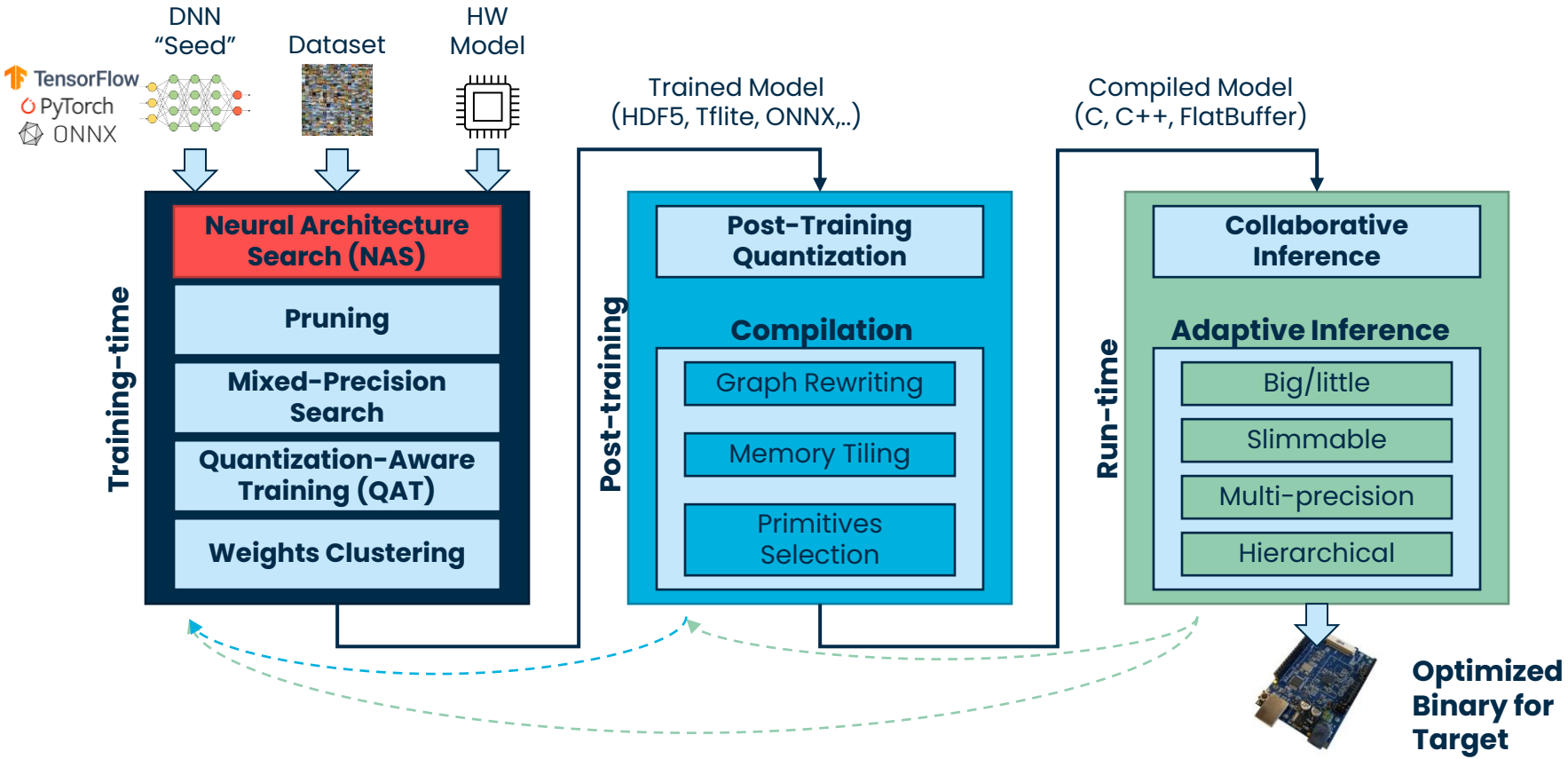
- **Adaptive/Dynamic Inference:**
  - Modify the operations executed by the model at runtime, adapting to:
    - The time-varying "difficulty" of processed data
    - The time-varying context (e.g., battery state)



Politecnico
di Torino

Neural Architecture Search (NAS)

DNN Deployment Flow



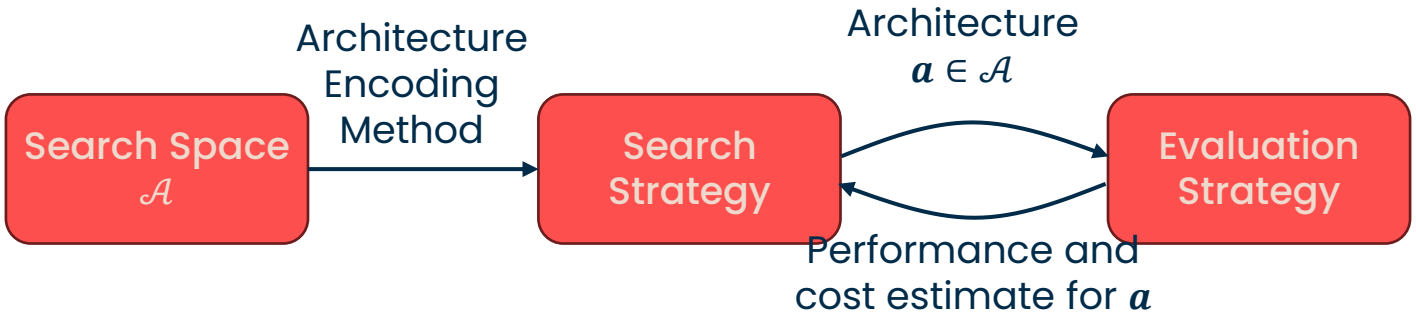
Neural Architecture Search

- Picking hyper-parameters manually is tricky:
  - Biases (rules of thumb, traditions, etc.)
  - Fragmented and coarse design space explorations (e.g., width/res mult in MobileNets)
  - **Classic ML: hand-craft features, DL: hand-craft feature extractors**!

- Neural Architecture Search (NAS):
  - Automatic optimization of the network topology, exploring a large and fine-grain design space of hyper-parameter settings
  - Typically **multi-objective**: co-optimize accuracy and model complexity
    - Model size (memory)
    - N. of Ops …or better, **latency/energy** (requires models)!

Neural Architecture Search (NAS)

- NAS Methods can be categorized according to:
  - **The search space:** what types architectural parameters are explored
  - **The search strategy:** what algorithm is used to find "good" architectures
  - **The evaluation strategy:** how are candidate architectures assessed



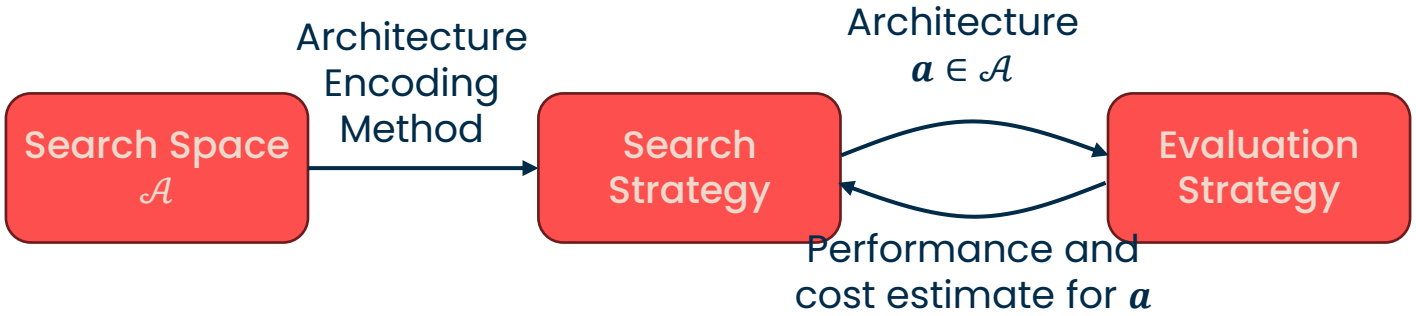
[source] C. White et al, "Neural Architecture Search: Insights from 1000 Papers", arXiv 2023

Neural Architecture Search (NAS)

- Basic NAS Objective (bi-level optimization):

$$\min_{\boldsymbol{a}\in\mathcal{A}} \mathcal{L}_{val}(\boldsymbol{w}^*(\boldsymbol{a}), \boldsymbol{a}) \quad \text{s.t.} \quad \boldsymbol{w}^*(\boldsymbol{a}) = \underset{w}{\operatorname{argmin}}\, \mathcal{L}_{train}(\boldsymbol{w}, \boldsymbol{a})$$

- This only targets accuracy. Non-functional metrics can be optionally added (see later)



[source] C. White et al, "Neural Architecture Search: Insights from 1000 Papers", arXiv 2023

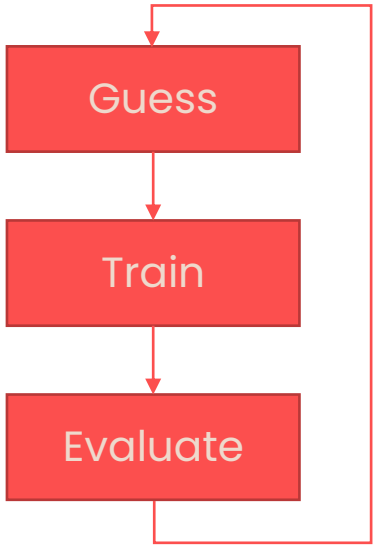
Neural Architecture Search (NAS)

- Three main families of search strategies:
  - **Classic Iterative NAS:** uses black-box methods such as Reinforcement Learning, Evolutionary Algorithms, Bayesian Optimization
  - **Differentiable NAS (DNAS):** uses back-propagation for simultaneous weights training and architecture optimization.
  - **One-shot/Weight-sharing NAS:** aimed at avoiding multiple searches on the same space.
    - E.g., when targeting multiple devices with different hardware constraints.
    - Can use either RL/EA/BO or gradient descent.

Classic NAS

- Procedure:

Propose 1+ new architecture(s)



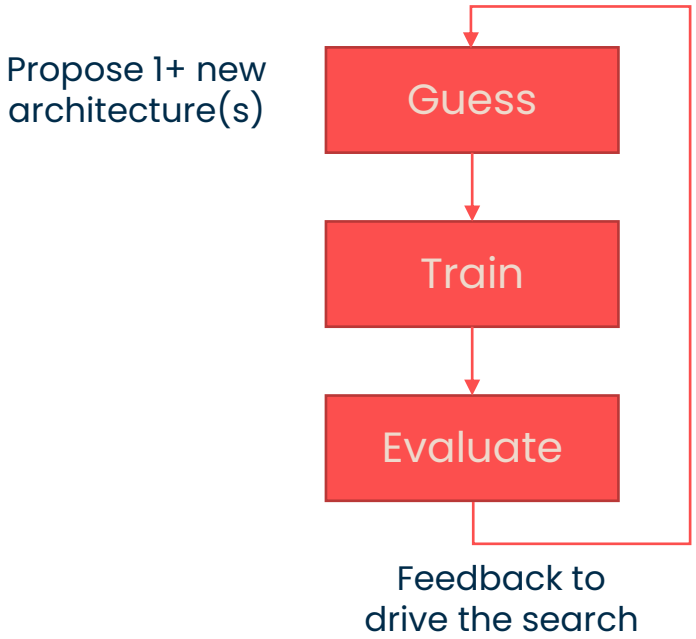
Feedback to drive the search

- Search engine:
  - Method used to “guess” next candidate(s)
  - Reinforcement Learning (RL)
  - Evolutionary Algorithm (EA)
  - Bayesian Optimization (BO)
  - Others…

- **Treats problem as a black-box.**

Classic NAS

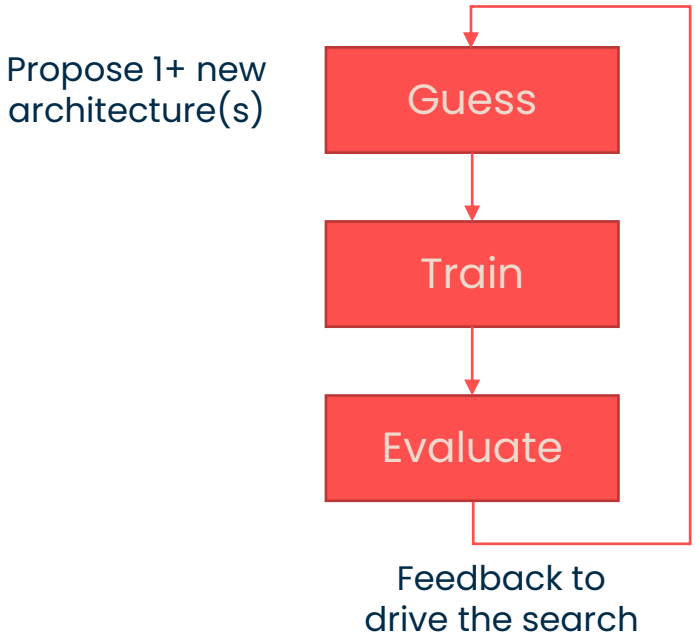
- Procedure:



- Evaluation Strategy
  - Estimating **accuracy**:
    - Encompasses **training**
  - Estimating **extra-functional "cost" metrics**:
    - Naïve solution: **deploy** every candidate
    - Alternative: accurate model.
- Huge bottleneck!
- **Thousands of GPU-hours per search!**

Classic NAS

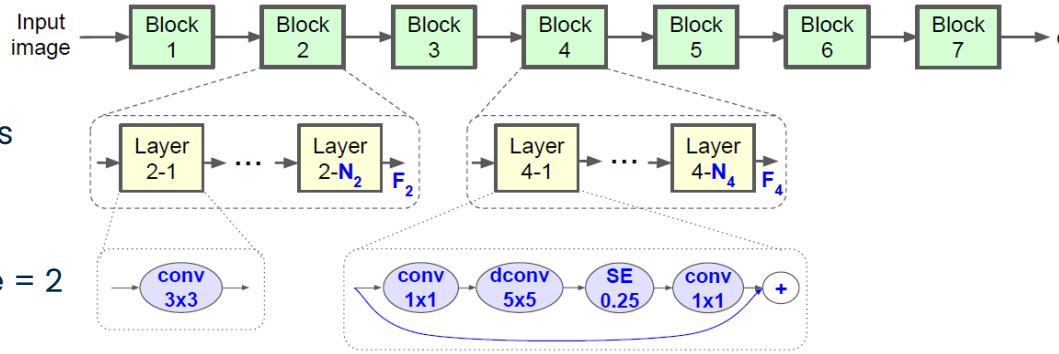
- Procedure:



- Search space:
  - Highly flexible w.r.t. optimized variables
    - **Number of layers**
    - **Layer type** (e.g. Conv vs DW)
    - **Layer hyper-parameters** (e.g., n. of channels, filter size, …)
    - Etc.
  - **But evaluation bottleneck forces search space size restrictions. For instance:**
    - **Discretize** each variable (e.g., K in {32, 64, 128}, FX=FY in {3, 5, 7}, etc.)
    - **Cell-based search space** (repeat identical cells)

Example of (Edge-oriented) Classic NAS: MNASNet

1. Search Space:
   - Fixed number of blocks (cell-based)
     - different input resolution & n. channels
   - N_i identical layers in each block
     - 1st layer in each block may have stride = 2
   - Optimized variables:
     - **Number of Layers** within each block (N_i)
     - **Layer operation**: Conv, DW+PW, Inverted Bottleneck
     - **Kernel size** ($FX * FY$): *3x3 or 5x5*
     - **Skip operation:** pooling, identity, no skip
     - **Squeeze and Excitation** (SE) Ratio: 0 (absent), 0.25
     - **Number of output channels** (K_i): 0.75, 1.00 or 1.25 times the MobileNetV2 value



[source] Tan et al, MnasNet: Platform-Aware Neural Architecture Search for Mobile, CVPR 2019

Example of (Edge-oriented) Classic NAS: MNASNet

2. Search Engine
   - Reinforcement learning with a Recurrent Neural Network as "controller"
   - Mobile-oriented → Latency-Constrained objective:

$$\max_{\boldsymbol{a}\in\mathcal{A}} Acc_{val}(\boldsymbol{w}^*(\boldsymbol{a}),\boldsymbol{a}) \quad s.t.\ Lat(\boldsymbol{a}) < T$$

Transformed to Reward

$$R(\boldsymbol{a}) = Acc_{val}(\boldsymbol{w}^*(\boldsymbol{a}),\boldsymbol{a}) \times \left[\frac{Lat(\boldsymbol{a})}{T}\right]^w$$

w < 0 and application-specific. w=-0.07 in the paper.

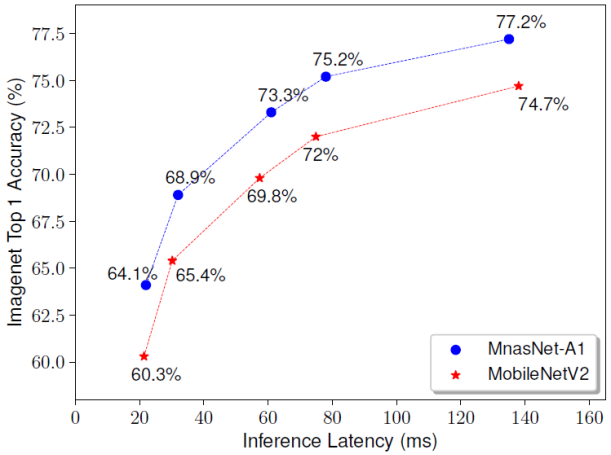
3. Evaluation Strategy
   - For each sample
     - Train
     - Deploy on HW
     - Measure Latency

\+ Very good estimate of both accuracy and latency

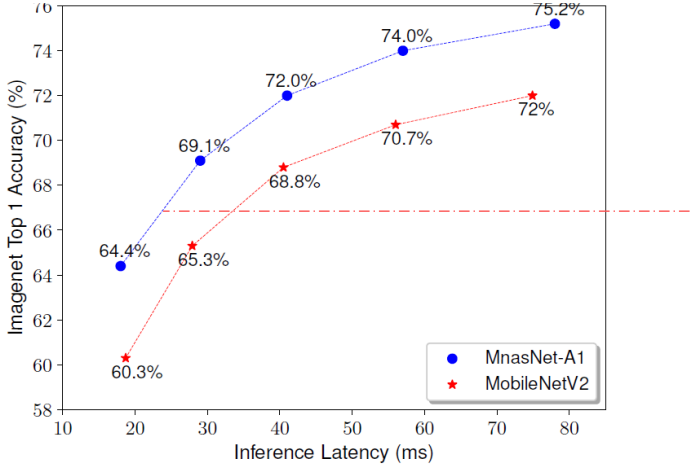
\− Huge cost: 4.5 days per single search on 64 TPUv2 devices

Found DNNs that Pareto-dominate the best MobileNets (at the time)

Mnas-Net: Results



(a) Depth multiplier = 0.35, 0.5, 0.75, 1.0, 1.4, corresponding to points from left to right.



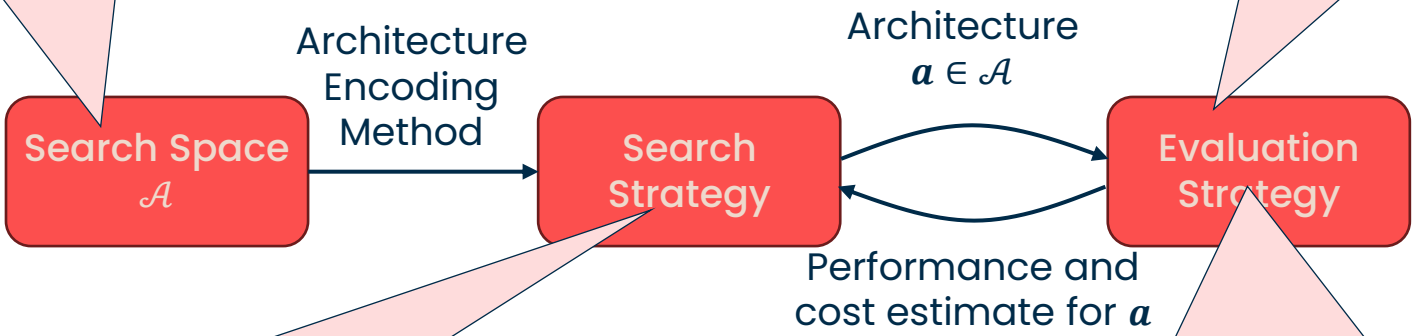
(b) Input size = 96, 128, 160, 192, 224, corresponding to points from left to right.

But each architecture search takes 4.5 days on 64 TPUv2 devices!!

Speeding Up

Constrain the Search Space:
- Remove/freeze "weak" variables
- Coarse discretization
- Cell-based/hierarchical search space

Avoid complete training:
- Train on a reduced data-set
- Train for few epochs (learning curve extrapolation)
- Avoid training entirely ("zero-cost" proxies)
- One-shot methods



Avoid RL or other complex optimizers:
- Use gradient descent (DNAS)

Avoid deployment:
- HW emulation/simulation
- HW modelling.

Differentiable NAS (DNAS)

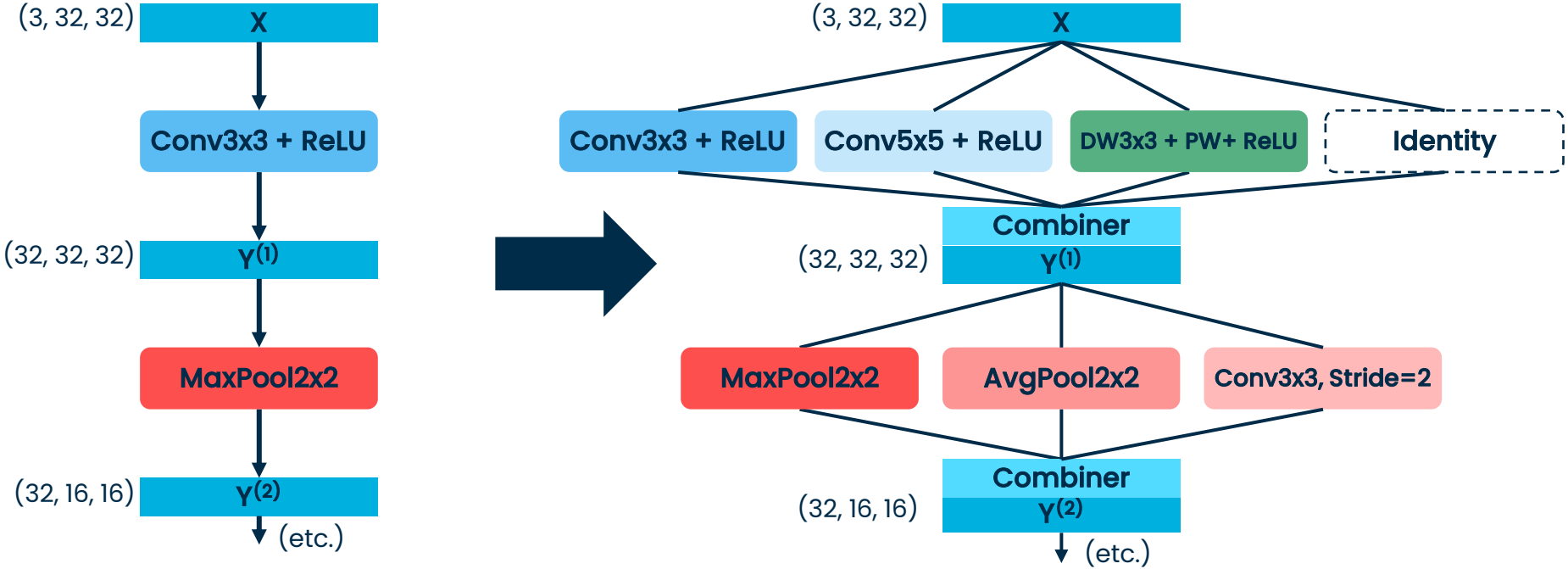
- **Relax the search space to make it continuous and differentiable**

- **Optimize the topology by gradient descent**
  - Reduce search costs…
  - Gradient-based optimization is much more lightweight than black-box methods (RL, EA, etc)

- **Early implementation:**
  - Liu et al, DARTS: Differentiable Architecture Search, ICLR 2019

SuperNet-based DNAS

1. Search Space: the “SuperNet”
   - **SuperNet =** DNN with multiple alternative operators. For example:

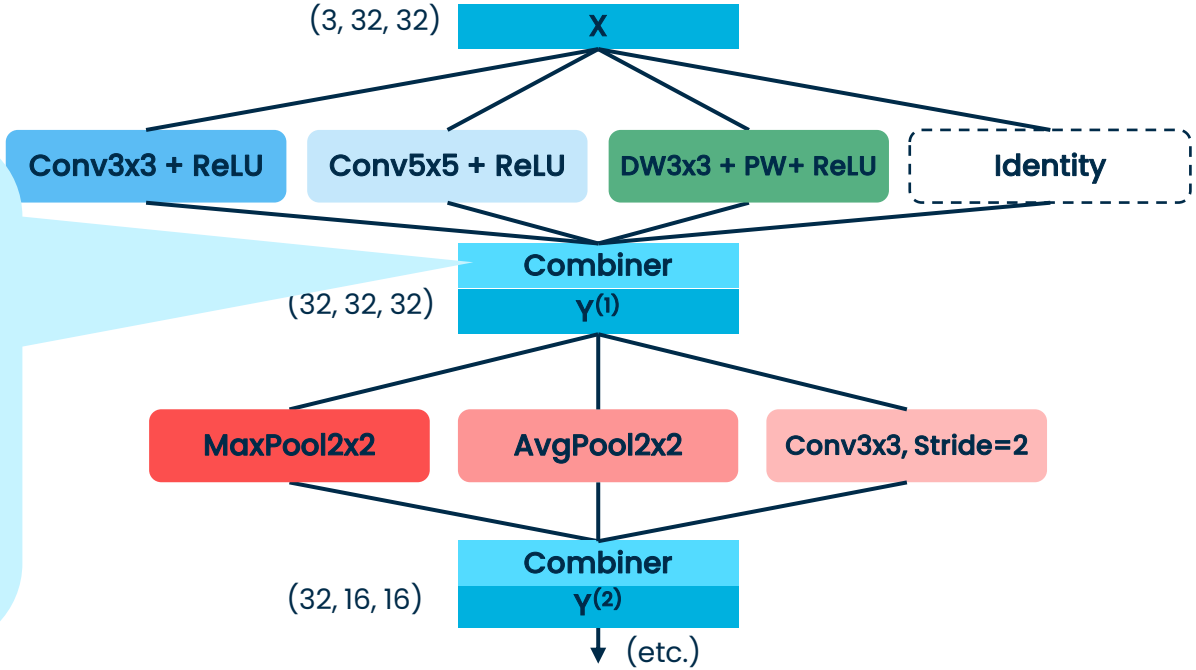


SuperNet-based DNAS

1. Search Space: the “SuperNet”

$$Y^{(1)} = \sum_i \theta_i^{(1)} Y_i^{(1)}$$

$$Y_0^{(1)} = ReLU(\text{Conv3}x3(X))$$
$$Y_1^{(1)} = ReLU(\text{Conv5}x5(X))$$
$$Y_2^{(1)} = ReLU(PW(DW(X)))$$
$$Y_3^{(1)} = X$$

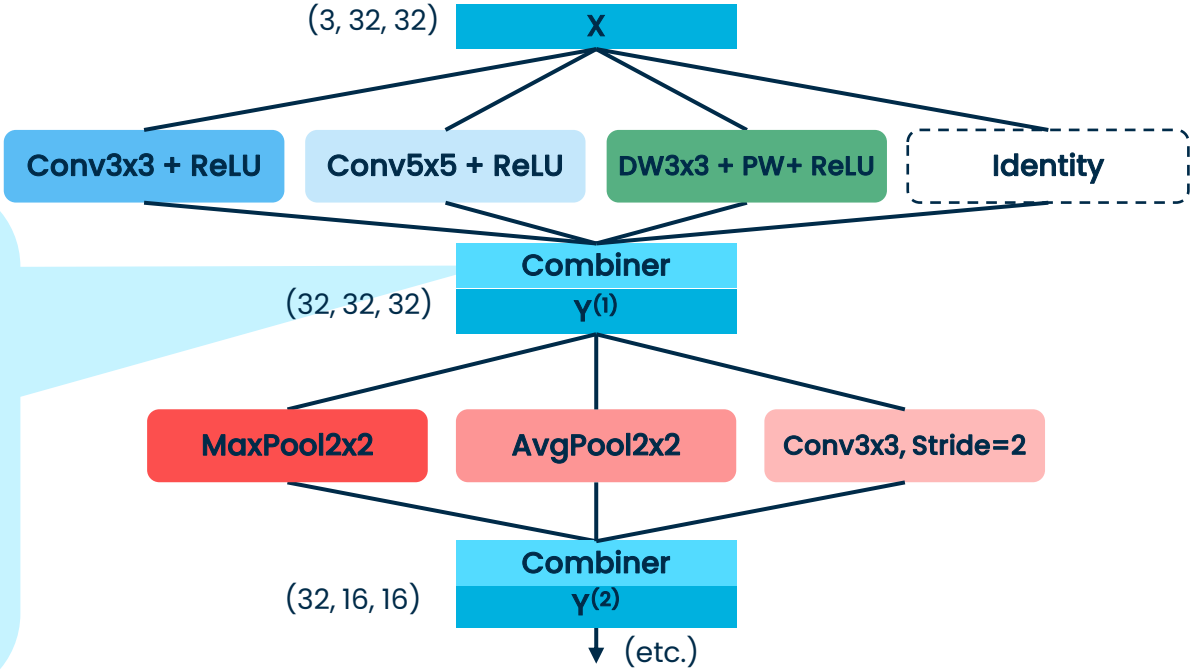


SuperNet-based DNAS

1. Search Space: the "SuperNet"

$$Y^{(1)} = \sum_i \theta_i^{(1)} Y_i^{(1)}$$

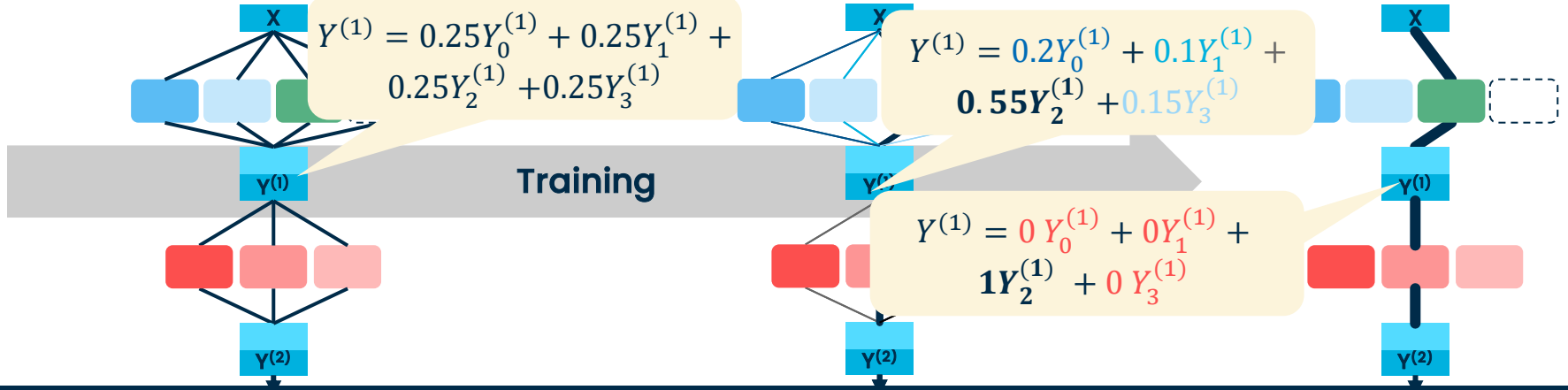
- **DNAS Objective:** set $\theta_i^{(l)} = 1$ for the ***"best"*** candidate Op. and $\theta_i^{(l)} = 0$ for all the others.
- Train $\theta_i^{(l)}$ together with the normal network weights W



SuperNet-based DNAS

1. Search Space: the "SuperNet"
   - Original DARTS paper approach:
     - Initially relax the choice using a **softmax**: $\theta_i^{(l)} = e^{\alpha_i^{(l)}} / \sum_i e^{\alpha_i^{(l)}}$
     - Train the $\alpha_i^{(l)}$ as the other network parameters.
     - At the end of training, discretize with an **argmax**

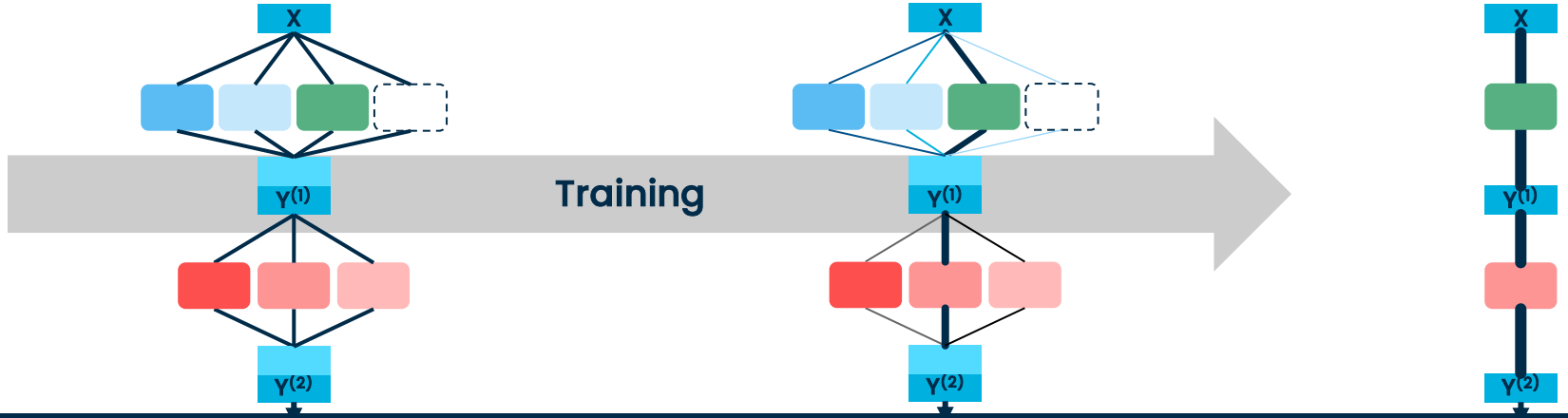
Argmax



SuperNet-based DNAS

1. Search Space: the "SuperNet"
   - Original DARTS paper approach:
     - Initially relax the choice using a **softmax**: $\theta_i^{(l)} = e^{\alpha_i^{(l)}} / \sum_i e^{\alpha_i^{(l)}}$
     - Train the $\alpha_i^{(l)}$ as the other network parameters.
     - At the end of training, discretize with an **argmax**

Final Model



DNAS Loss Function

- If **goal = max. accuracy**, the training loss is the NAS objective seen before
  - Where the architecture a is now encoded by trainable parameters $\boldsymbol{\alpha}$:

$$\min_{\boldsymbol{\alpha}} \mathcal{L}_{val}(\boldsymbol{w}^*(\boldsymbol{\alpha}), \boldsymbol{\alpha}) \quad \text{s.t.} \quad \boldsymbol{w}^*(\boldsymbol{\alpha}) = \operatorname{argmin}_{\boldsymbol{w}} \mathcal{L}_{train}(\boldsymbol{w}, \boldsymbol{\alpha})$$

- Additional non-functional objectives can be added (next slide)

- Practical approximation:
  - To reduce training costs, in DARTS, **w*** are not the weights at convergence, but the results of a "few" training steps (e.g. 1 minibatch, 1 epoch,…)

Cost-aware DNAS

- **The simplest way to consider cost metrics in DNAS consists in adding a regularization term $\mathcal{R}(\alpha)$ to the loss function**
  - This term should encourage gradient-descent to prefer NAS parameters (α) that correspond to efficient models
  - It must model network "cost" (memory, FLOPs, latency, energy) in a **differentiable** way

- The complete loss function becomes:

$$\boldsymbol{min}[\,\mathcal{L}(\boldsymbol{w},\boldsymbol{\alpha}) + \lambda\mathcal{R}(\boldsymbol{\alpha})]$$

  - λ is a scalar **regularization strength** (discussed later)

Cost-aware DNAS

- Example with model size (i.e. weights memory) as cost metric:

- Regularizer: R
  - Where $R^{(l)}$ is the e… size for layer l

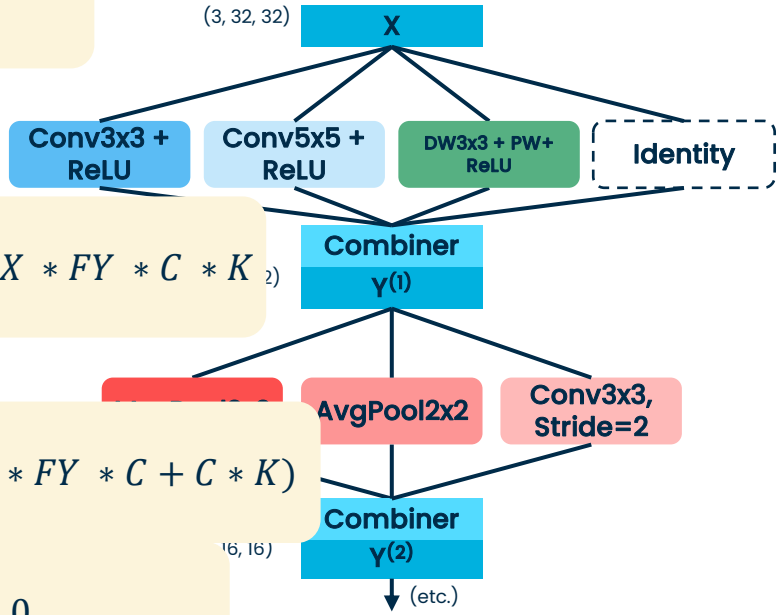
  - For instan…
    $R^{(1)}(\alpha^{(\cdot)}) =$
    $\theta_0^{(1)} * \mathbf{3} * \mathbf{3} * 3 * 32 +$
    $\theta_1^{(1)} * \mathbf{5} * \mathbf{5} * 3 * 32 +$
    $\theta_2^{(1)} * (3 * 3 * 3 + 3 * 32) +$
    $\theta_3^{(1)} * 0$

Remember that $\theta^{(l)} = f(\alpha^{(l)})$

Size of Standard Conv: $FX * FY * C * K$

Size of DW + PW: $(FX * FY * C + C * K)$

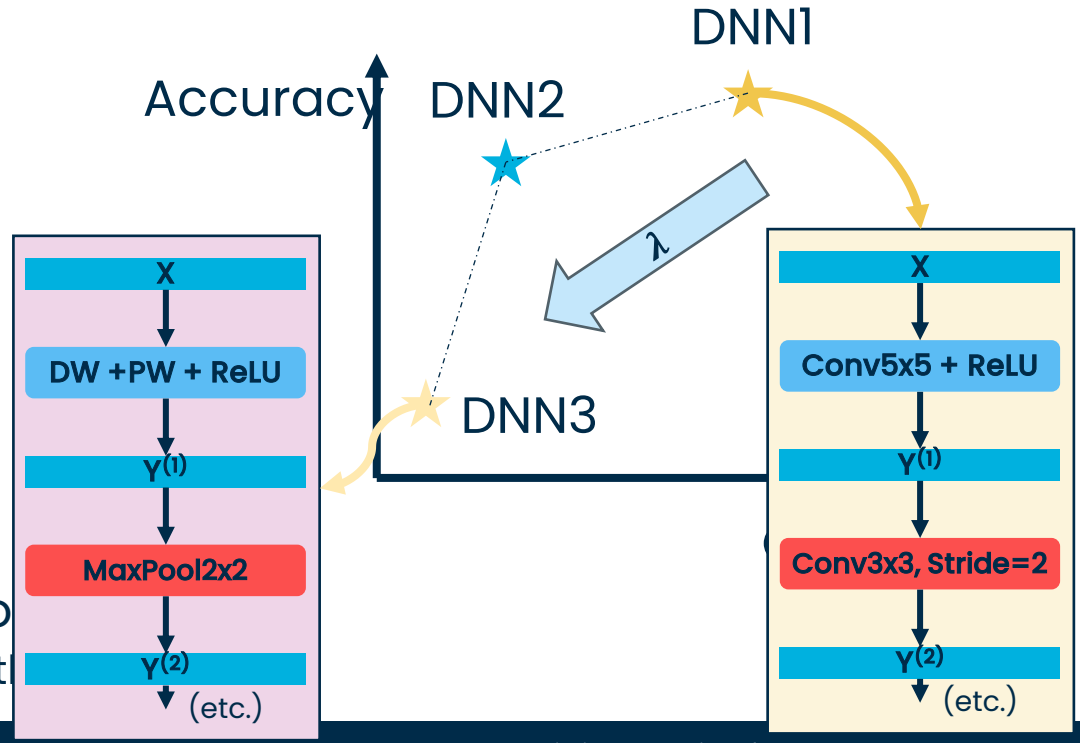
Size of "identity": 0



Cost-Aware DNAS

- Changing the reg. strength λ yields different trade-offs between accuracy and cost

$$\min[\mathcal{L}(w,\alpha) + \lambda\mathcal{R}(\alpha)]$$



- Many variants of this basic fo
  - Depending on the paper/algorit

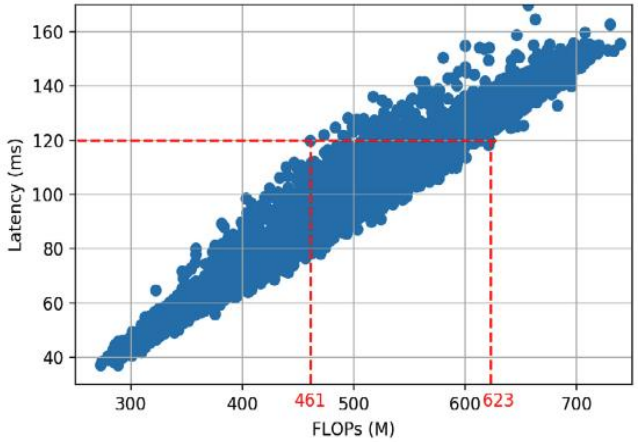
DNAS versus Classic NAS

DNAS produces one optimized architecture in a single SuperNet training.

- Search time orders of magnitude faster than a classic iterative approach.
- But the SuperNet is still larger than a “normal” DNN

DNAS Enhancements

- Reducing rank disorder:
  - SuperNet "sub-networks" may not perform consistently when isolated (co-adaptation)
  - Solutions include using a noisy and/or discrete sampling
    - E.g., Gumbel SoftMax possibly with a straight—through estimator (see later)
- Advanced cost models:
  - Modeling **Size (or FLOPs)** in a differentiable way is easy
  - **FLOPs correlate with latency and energy, but they are not the same thing!**
    - Latency depends on memory hierarchy, HW parallelism, etc.
  - One solution is to build a DNN-based (thus **differentiable**) latency/energy predictor.

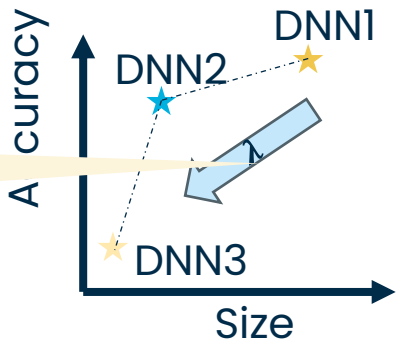


[source] D. Liu et al, "Bringing AI to edge: From deep learning's perspective", Neurocomputing, 2022.

DNAS Enhancements

- Constrained Optimization:
  - The $\boldsymbol{min}[\,\mathcal{L}(\boldsymbol{w},\boldsymbol{\alpha}) + \boldsymbol{\lambda}\,\boldsymbol{R}(\boldsymbol{\alpha})]$ formulation does not allow imposing **cost contraints**.
    - E.g. we may want to find the **most accurate model with $size < S$**
  - A possible solution is to penalize the loss only if the constraint is exceeded. E.g.: $\boldsymbol{min}[\,\mathcal{L}(\boldsymbol{w},\boldsymbol{\alpha}) + \boldsymbol{\lambda}\,\boldsymbol{max}(\boldsymbol{R}(\boldsymbol{\alpha}) - \boldsymbol{S}, \boldsymbol{0})]$
    - *A. Burrello et al, Enhancing neural architecture search with multiple hardware constraints for deep learning model deployment on tiny iot devices, IEEE TETC 2023.*
- SuperNet training efficiency:
  - The SuperNet is bulky to train.
    - Solutions "sample" one or few paths and only load those during training
    - *H. Cai et al, ProxylessNAS: Direct Neural Architecture Search on Target Task and Hardware, ICLR 2019*
  - With DARTS-like approaches, each Pareto-optimal DNN requires a separate SuperNet training
    - **Solution: one-shot NAS**

Train SuperNet 3 times

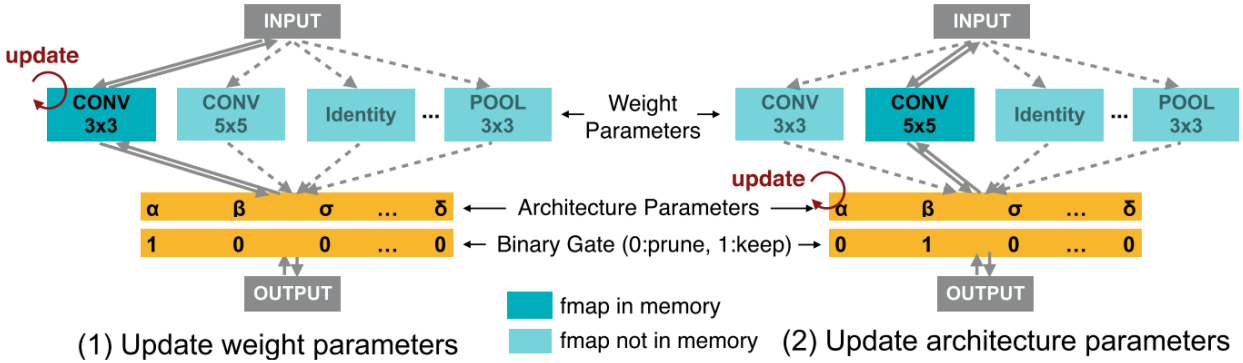


Advanced DNAS Example: ProxylessNAS

- ProxylessNAS builds upon the DNAS idea, but trains a **single SuperNet path** at a time (sampling based on the current probabilities):

$$m_{\mathcal{O}}^{\text{DARTS}}(x) = \sum_{i=1}^{N} p_i o_i(x) = \sum_{i=1}^{N} \frac{\exp(\alpha_i)}{\sum_j \exp(\alpha_j)} o_i(x).$$

$$m_{\mathcal{O}}^{\text{Binary}}(x) = \sum_{i=1}^{N} g_i o_i(x) = \begin{cases} o_1(x) & \text{with probability } p_1 \\ \cdots & \\ o_N(x) & \text{with probability } p_N. \end{cases}$$



[Source] H. Cai et al, ProxylessNAS: Direct Neural Architecture Search on Target Task and Hardware, 2019

Advanced DNAS Example: ProxylessNAS

- Uses the regularization loss described before:
  - Directly builds differentiable model for energy/latency
  - Profile many layers configurations and build a linear/polynomial regression model: Latency = F(Layer Params)

$$\mathbb{E}[\text{latency}_i] = \sum_j p_j^i \times F(o_j^i),$$

$$Loss = Loss_{CE} + \lambda_1 ||w||_2^2 + \lambda_2 \mathbb{E}[\text{latency}]$$

NAS Parameter (prob. of o^i_j)

Latency Model

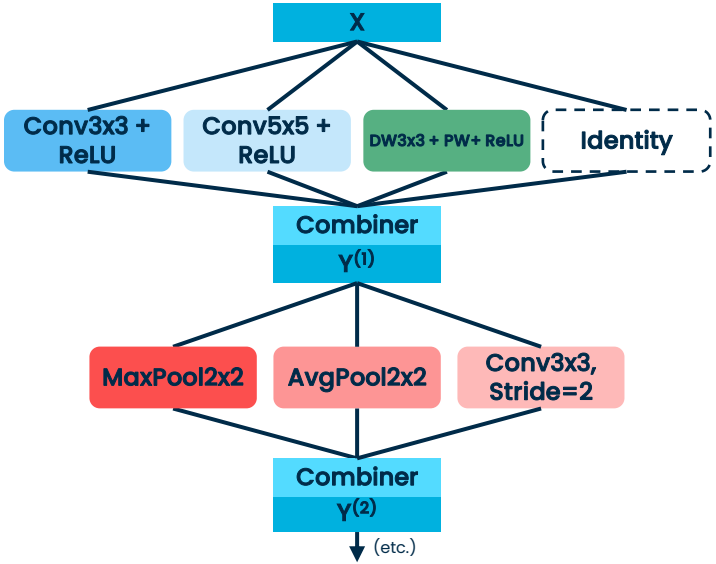
Standard weight decay

Advanced DNAS Example: ProxylessNAS

Model	Top-1	Top-5	Mobile Latency	Hardware -aware	No Proxy	No Repeat	Search cost (GPU hours)
MobileNetV1 [16]	70.6	89.5	113ms	-	-	✗	Manual
MobileNetV2 [30]	72.0	91.0	75ms	-	-	✗	Manual
NASNet-A [38]	74.0	91.3	183ms	✗	✗	✗	48, 000
AmoebaNet-A [29]	74.5	92.0	190ms	✗	✗	✗	75, 600
MnasNet [31]	74.0	91.8	76ms	✓	✗	✗	40, 000
MnasNet (our impl.)	74.0	91.8	79ms	✓	✗	✗	40, 000
Proxyless-G (mobile)	71.8	90.3	83ms	✗	✓	✓	200
Proxyless-G + LL	74.2	91.7	79ms	✓	✓	✓	200
Proxyless-R (mobile)	**74.6**	**92.2**	78ms	✓	✓	✓	200

One-shot NAS (Short)

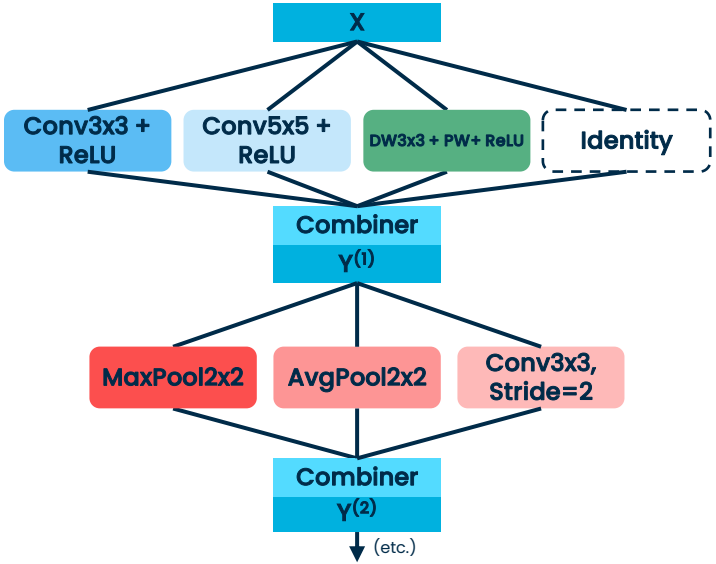
- **Train the SuperNet once-for-all.**
  - Keeping **all paths active** with equal θ values, or…
  - …discretely sampling 1 or more paths in each iteration.
- After training, multiple single-path networks can be "extracted" from the SuperNet:
  - Both **both black-box or gradient-based** search engines can be used.
- **No further training to evaluate candidates' accuracy.**
  - Only testing on a validation set. This **assumes rank order**…
  - At the end, the best candidate is usually **fine-tuned**.



One-shot NAS (Short)

- **Usage:**
  - Obtain multiple Pareto-optimal networks with a single SuperNet training
  - Example: targeting a range of devices with different characteristics.

- Representative Papers:
  - H. Cai et al, “Once-for-All: Train One Network and Specialize it for Efficient Deployment”, ICLR 2020
  - Z. Guo et al, “Single Path One-Shot Neural Architecture Search with Uniform Sampling”, ECCV 2020

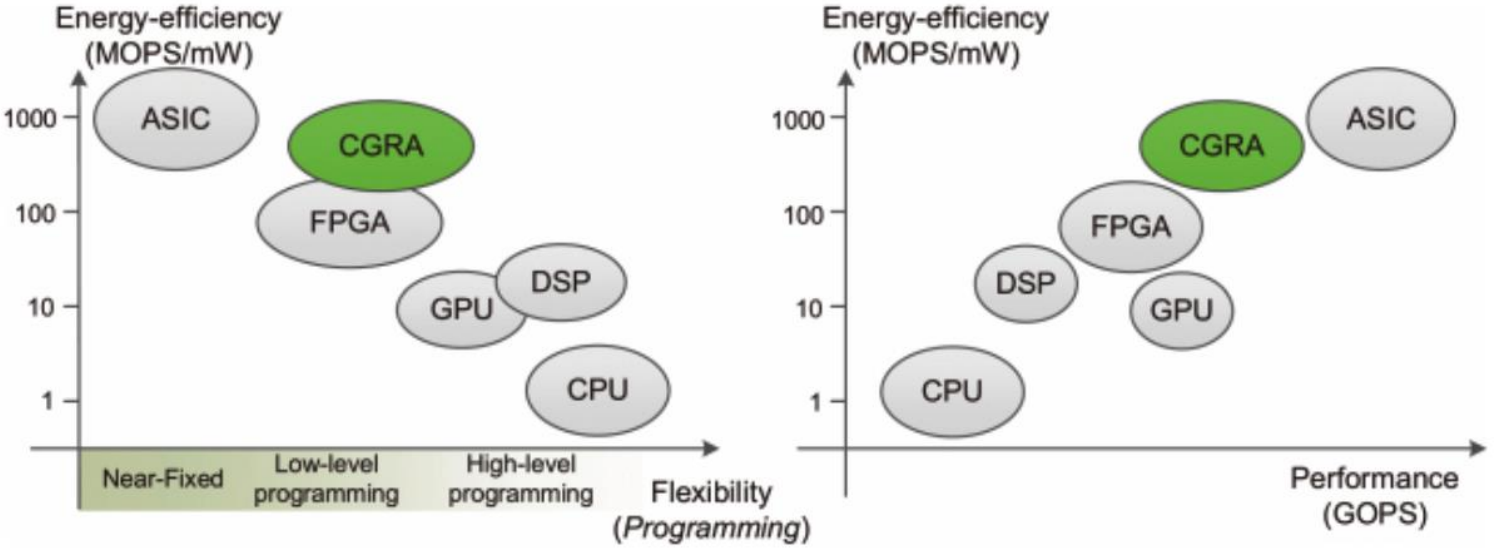


Politecnico
di Torino

Extra: Deep Learning Inference Hardware

Deep Learning Inference HW

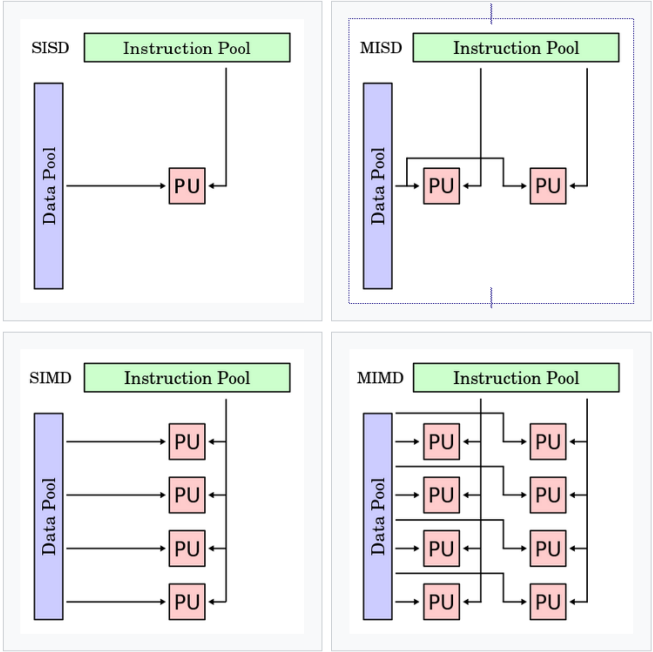
- HW devices for implementing deep learning inference:



Flynn Taxonomy

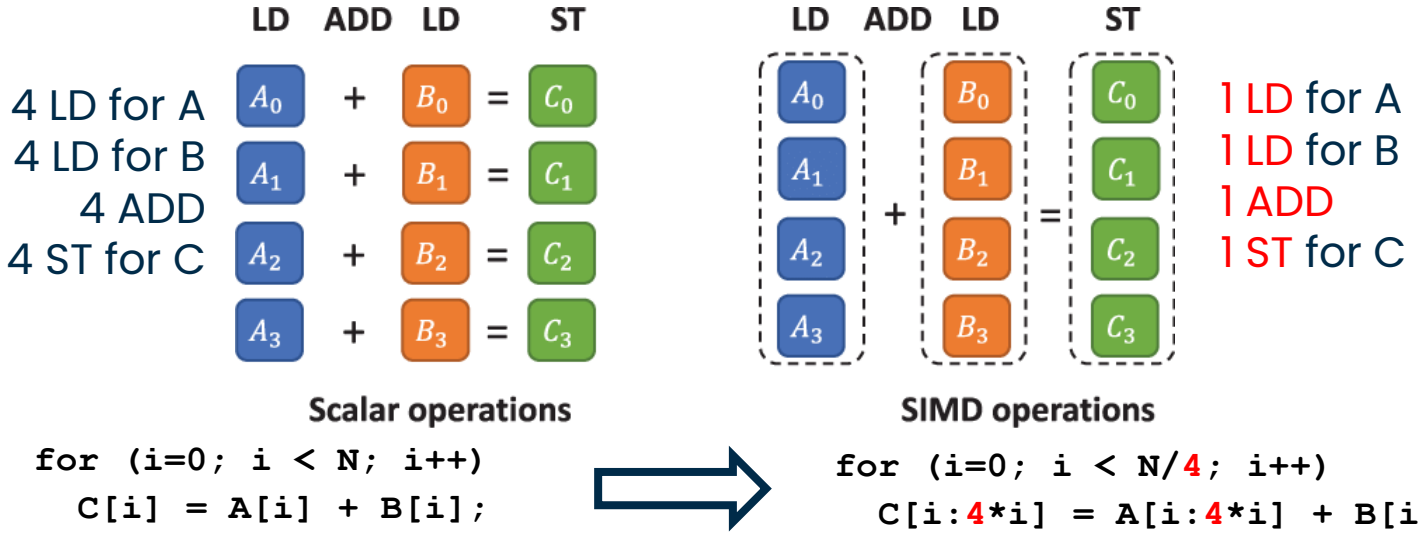
- Flynn Taxonomy (1966) classifies architectures based on the adopted parallel axis:
  - Instructions
  - Data

DATA \ INSTRUCTIONS	Single	Multiple
Single	**SISD** Single Instruction Single Data (Scalar Core)	**MISD** Redundant Processor
Multiple	**SIMD** Single Instruction Multiple Data (SIMD ISA Extension, Vector, GPGPU)	**MIMD** Multiple Instruction Multiple Data (Multiprocessors, Multithreading)



SISD vs SIMD

- SIMD exploits parallelism on different data: operation concurrently applied to different pieces of data




```
for (i=0; i < N; i++)
  C[i] = A[i] + B[i];
```

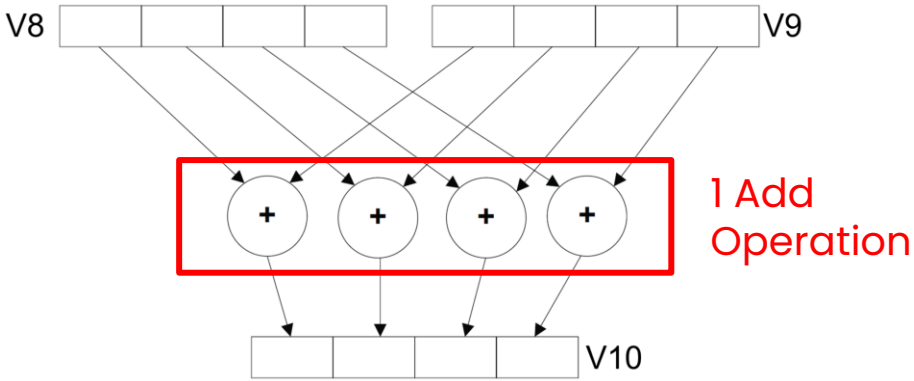
```
for (i=0; i < N/4; i++)
  C[i:4*i] = A[i:4*i] + B[i:4*i];
```

With a parallelism of 4 (memory access = 1/4)

SIMD ISA Extensions

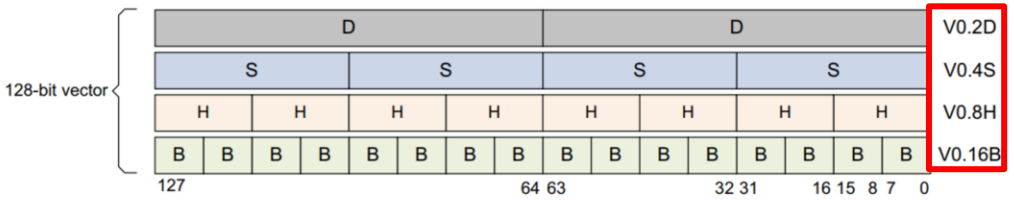
- Single Instruction Multiple Data (SIMD) extension instructions
  - Single instruction acts on multiple pieces of data at once
- Real Examples : NEON -> ARM ; SSE/AVX -> INTEL
- Ex. add four 8-bit numbers
  - 1 Register 32bits contains 4 8-bits numbers
  - The Adder is modified so that it performs element-wise addition (8bit, 16bit, 32bit OPs)

Vectorial Register



ARM AARCH64 NEON ISA Extension

- NEON is the SIMD Technology introduced by ARM
- SIMD exposed at the ISA level
  - Registers
  - Instructions
- Vectorial Register File
  - 32 Register of 128-bits
  - Each register can be interpreted as
    - 2 64-bits elements
    - 4 32-bits elements
    - 8 8-bits elements



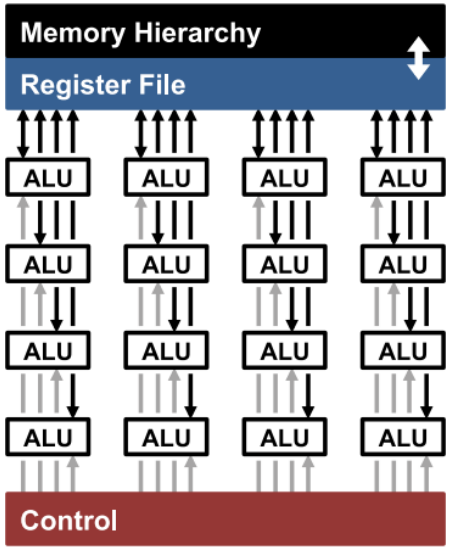
Same Register Name, but different suffix

Deep Learning Inference HW

- Temporal Architectures:
  - Multi/many core with **Single Instruction Multiple Data** (SIMD) extensions
  - **Single Instruction Multiple-Thread** (SIMT), e.g. GPUs
- Spatial Architectures:
  - **Dataflow/Systolic** (TPUs, NPUs)

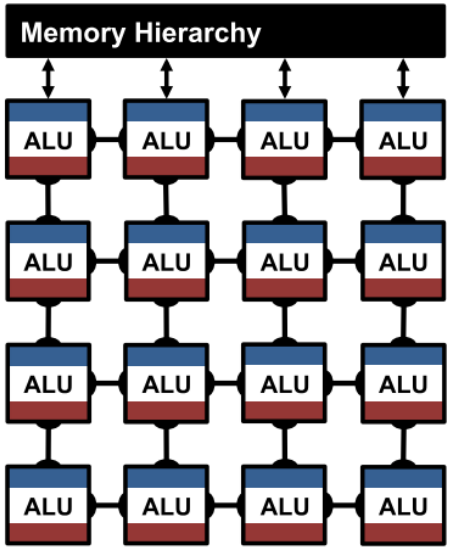
Deep Learing Inference HW

Temporal Architecture (SIMD/SIMT)



CPU/GPU

Spatial Architecture (Dataflow Processing)

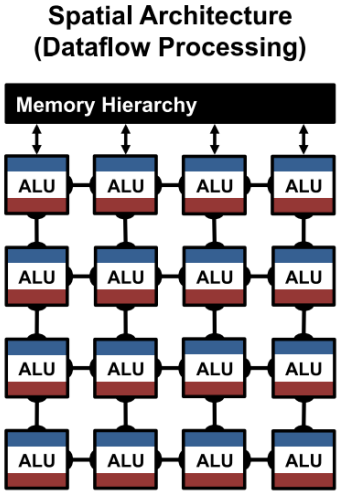


ASIC/FPGA

[Source] Sze et al, Efficient Processing of Deep Neural Networks: A Tutorial and Survey

Deep Learning Inference HW

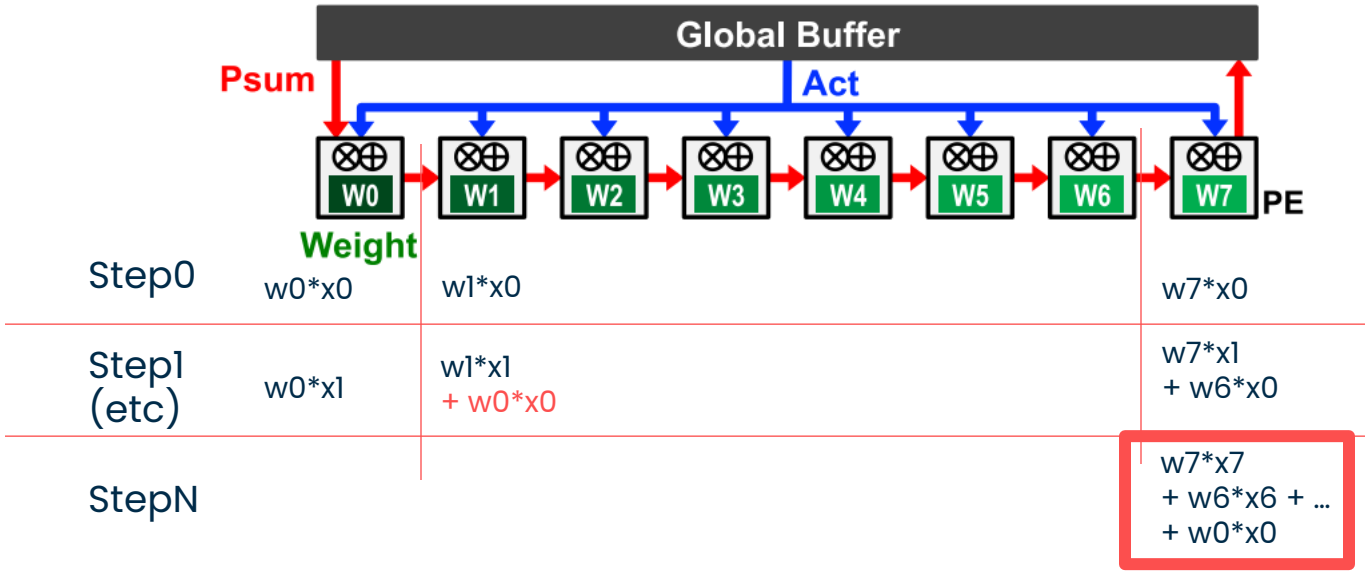
- ASICs for Deep Learning:
- Dataflow:
  - ALUs form a processing chain and can pass data from one to another directly.
  - Each ALU can have its own control logic and local memory (scratchpad)
  - ALU with its own local memory called processing element (PE).
  - Dataflow increases **data reuse** to reduce energy



ASIC/FPGA

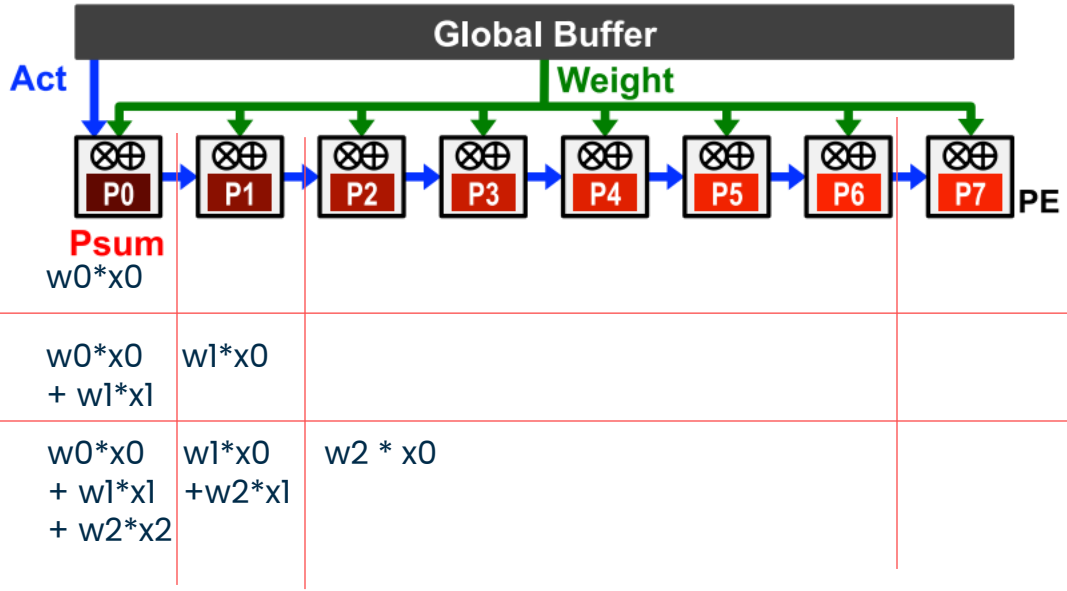
Deep Learning Inference HW

- ASICs for Deep Learning:
  - **Weight Stationary Dataflow** (Convolutional Reuse): each PE stores a weight
  - Input fmaps are broadcast to all PEs and then the partial sums are spatially accumulated across the PE array.



Deep Learning Inference HW

- ASICs for Deep Learning:
  - **Output Stationary Dataflow** : each PE stores a partial sum
  - Weights are broadcast to all PEs and inputs fmaps are streamed across the PE array.



Deep Learning Accelerators

- Two good surveys on energy-efficient edge deep learning:
  - V. Sze et al. *Efficient Processing of Deep Neural Networks: A Tutorial and Survey.* Proceedings of the IEEE, 2017. [HW Oriented]
  - J. Chen et al. *Deep Learning With Edge Computing: A Review*. Proceedings of the IEEE, 2019. [SW Oriented]

- Great blog post about HW accelerators on Medium:
  - Link

Rest of the Course

- In the rest of this course, we'll try to be as HW-agnostic as possible

- When we'll be HW specific, we'll mostly focus on General Purpose programmable devices (CPUs/GPUs)