# Learning to infer: RL-based search for DNN primitive selection on Heterogeneous Embedded Systems

Miguel de Prado, et al.

**DATE 2019**

**2019-12-11**

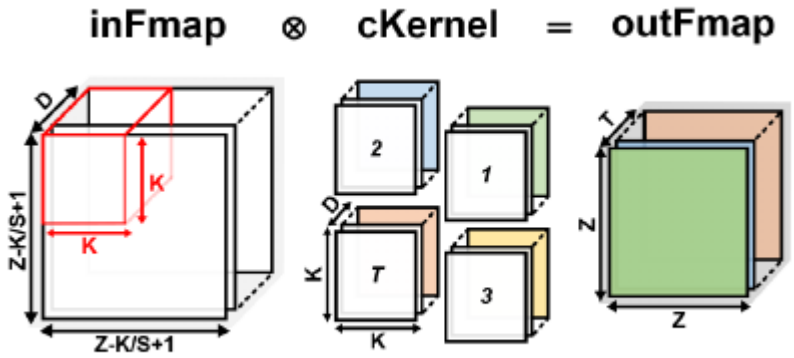
**Constant Park**

# Primitive Selection

- **Primitive: Method to compute convolution layer**
  - Multi channel and Multi kernel (MCMK)
  - General matrix multiplication (GEMM)
    - Image to column/row (Im2col/row), Kernel to column (Kn2col/row)
  - Fast Fourier transform (FFT)
  - Strassen, Winograd matrix multiplication

# Multi channel and Multi kernel (MCMK)

- **컨볼루션 레이어를 구현하는 가장 보편적인 구현 방법**
  - **Multiple for loop**을 사용하여 구현 (6~7 수준의 Nested for loop)
  - **Data locality**에서 가장 비효율적인 컨볼루션 가속 방법



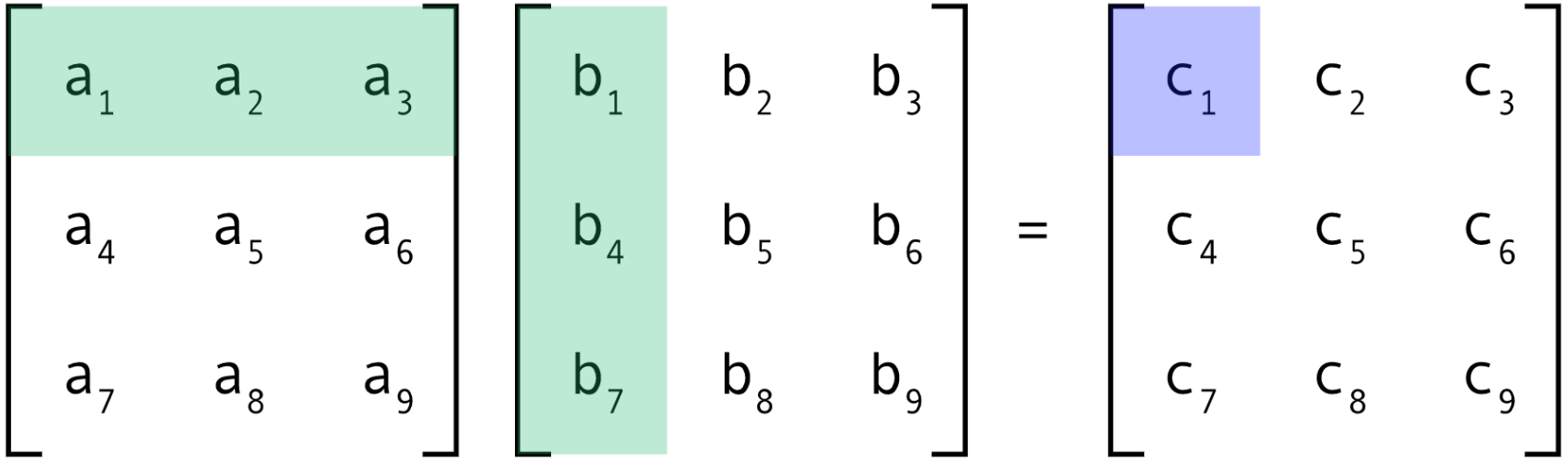

(a) Illustration of convolution layer

```
for (t=0; t<T; t++) {                    // type index of kernel
  for (r=0; r<Z; r=r+S) {                // row index of feature map
    for (c=0; c<Z; c=c+S) {              // col index of feature map
      for (d=0; c<D; d++) {              // depth index of feature map
        for (i=0; i<K; i++) {            // row index of kernel
          for (j=0; j<K; j++) {          // col index of kernel
           outFmap[t][r][c]+=inFmap[d][r+i][c+j]*cKernel[t][d][i][j]
      }}}
     outFmap [t][r][c]= activation(outFmap[t][r][c])}}}
```

(b) Pseudo code of convolution layer

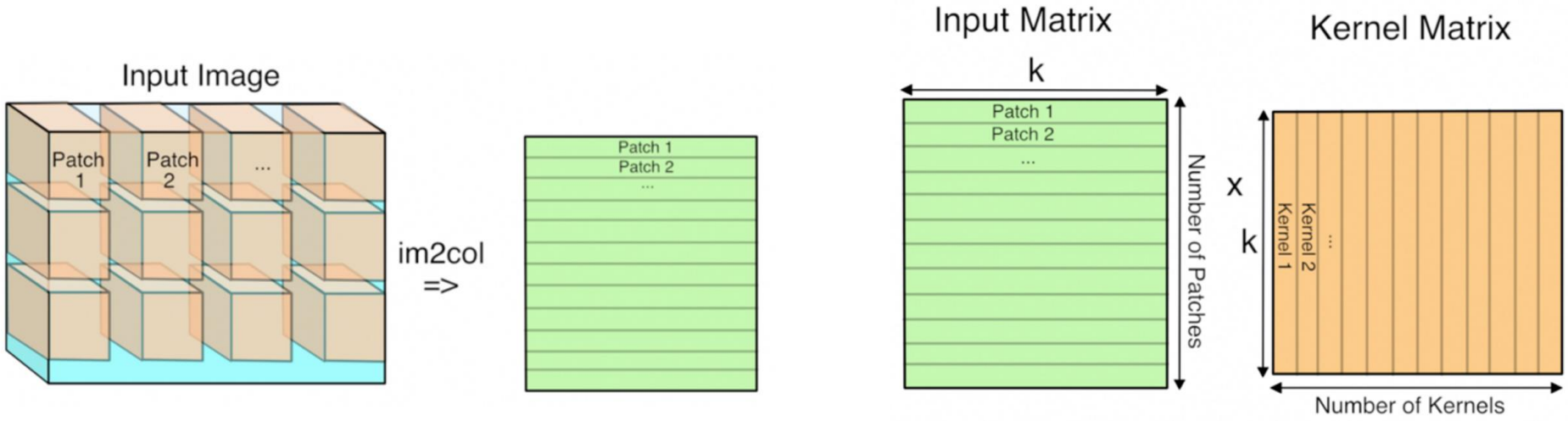
# General Matrix Multiplication (GEMM)

- **컨볼루션 레이어를 행렬로 변환하여 연산**
  - 행렬 연산을 위한 변환 과정, **행렬 연산**을 통해 결과를 얻음
  - MCMK보다 **Data locality에 유리한 방법**, 빠르게 가속 가능



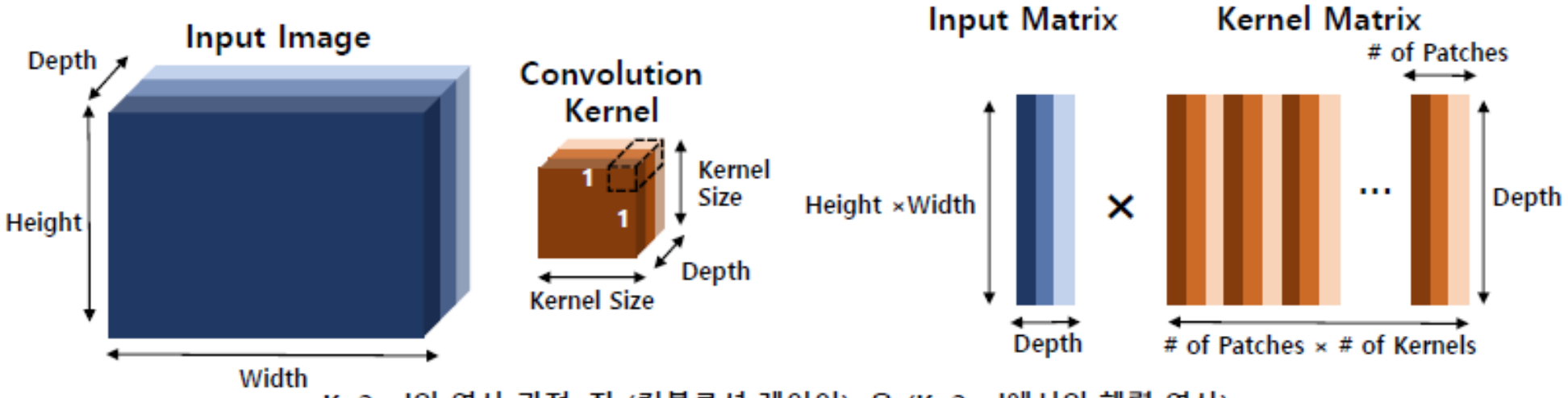
# Image to column/row (Im2col/row)

- **피쳐맵을 커널 사이즈에 대응할 수 있도록 패치로 변환**
  - **패치**: 커널이 이동하면서 feature map에 대응되는 부분 (receptive field)
  - 일반적으로 Im2col의 transpose된 Image to row (Im2row)가 사용 됨



# Kernel to column/row (Kn2col/row)

- **컨볼루션 커널을 패치로 변환**
  - 가로 (5) *세로 (5) * 깊이 (3) -> 가로 (1) * 세로 (1) * 깊이 (3) * 패치 개수 (25)
  - 1*1 컨볼루션에 적합한 방법




<Kn2col의 연산 과정, 좌 (컨볼루션 레이어), 우 (Kn2col에서의 행렬 연산)>

# FFT and Strassen/Winograd

- **행렬 곱셈에서 곱셈의 오버헤드를 개선하는 방법**
  - 곱셈은 덧셈에 비해 HW/SW 구현에서 더 많은 Overhead 존재
  - **곱셈 횟수를 줄이고 덧셈의 횟수를 늘리는 방법**
  - **곱셈을 Shift 연산으로 구현하는 방법**

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

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

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

$$Y = A^T\left[(Gg) \odot (B^T d)\right]$$

$$B^T = \begin{bmatrix} 1 & 0 & -1 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & -1 & 1 & 0 \\ 0 & 1 & 0 & -1 \end{bmatrix}$$

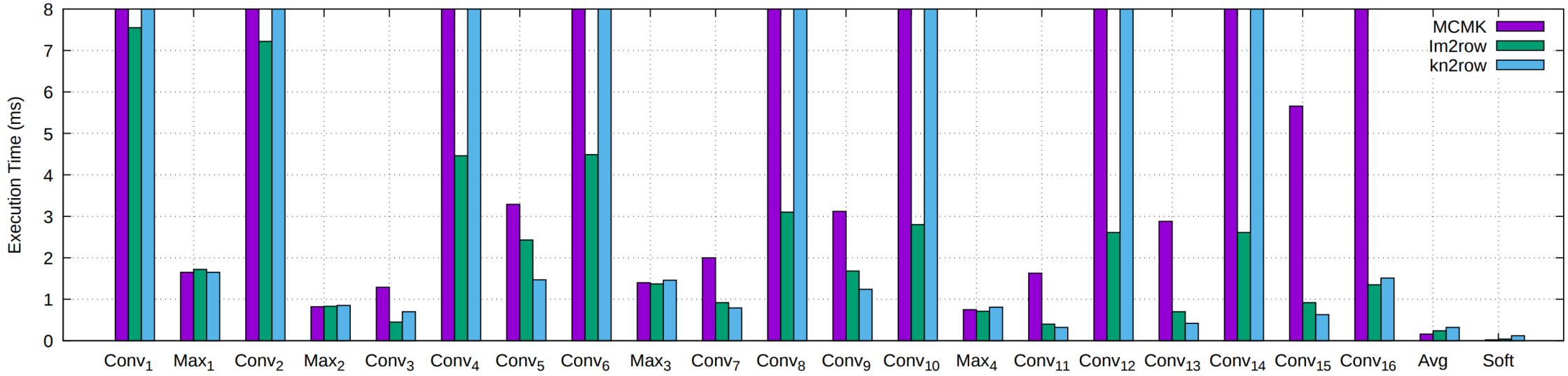
$$G = \begin{bmatrix} 1 & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & \frac{1}{2} \\ \frac{1}{2} & -\frac{1}{2} & \frac{1}{2} \\ 0 & 0 & 1 \end{bmatrix} \qquad A^T = \begin{bmatrix} 1 & 1 & 1 & 0 \\ 0 & 1 & -1 & -1 \end{bmatrix}$$

$$g = \begin{bmatrix} g_0 & g_1 & g_2 \end{bmatrix}^T$$

$$d = \begin{bmatrix} d_0 & d_1 & d_2 & d_3 \end{bmatrix}^T$$

# Why Primitive Selection ?

- **레이어의 특성에 따라 적합한 가속 방법이 상이**
  - Tiny darknet 신경망
  - $Conv_1$ (Im2row), $Conv_5$ (Kn2row)

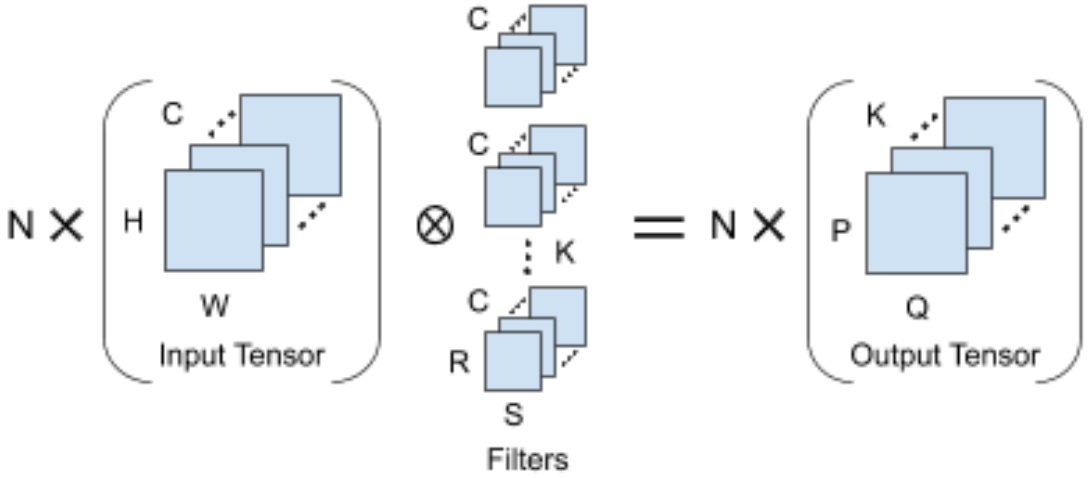


# ACCELERATION LIBRARIES

- Vanilla: CPU dependency-free and implemented in ANCI C
- Basic linear algebra subprograms (BLAS)
- NNPACK: low-level performance primitives on CPU core for DL
- ArmCL: optimized for Arm processor
- cuDNN, cuBLAS

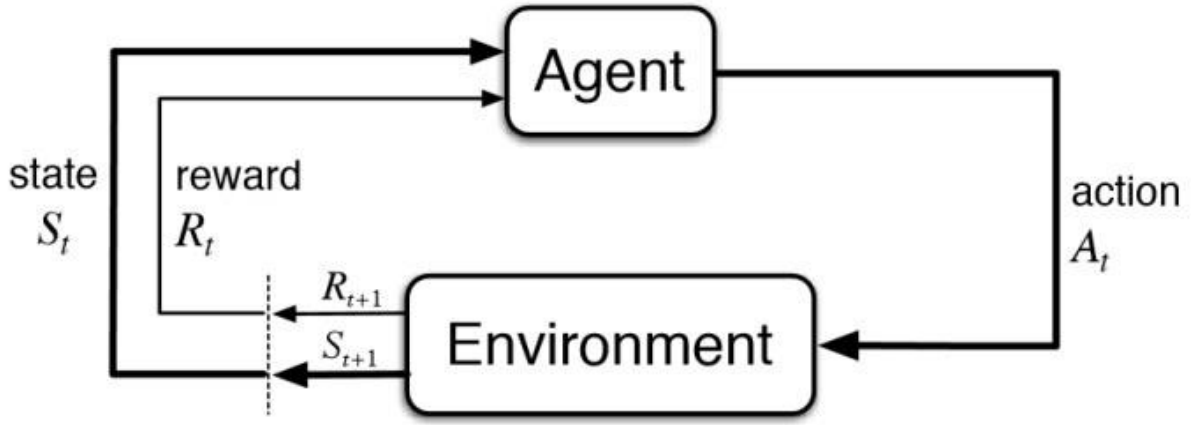
# Problem Formulation

- **Processing layer**
  - Executed by different acceleration libraries
- **Tensor of primitive**
  - Different input/output tensor layout (NCHW, WHNC)
- **Various design space**



# Reinforcement Learning (RL)

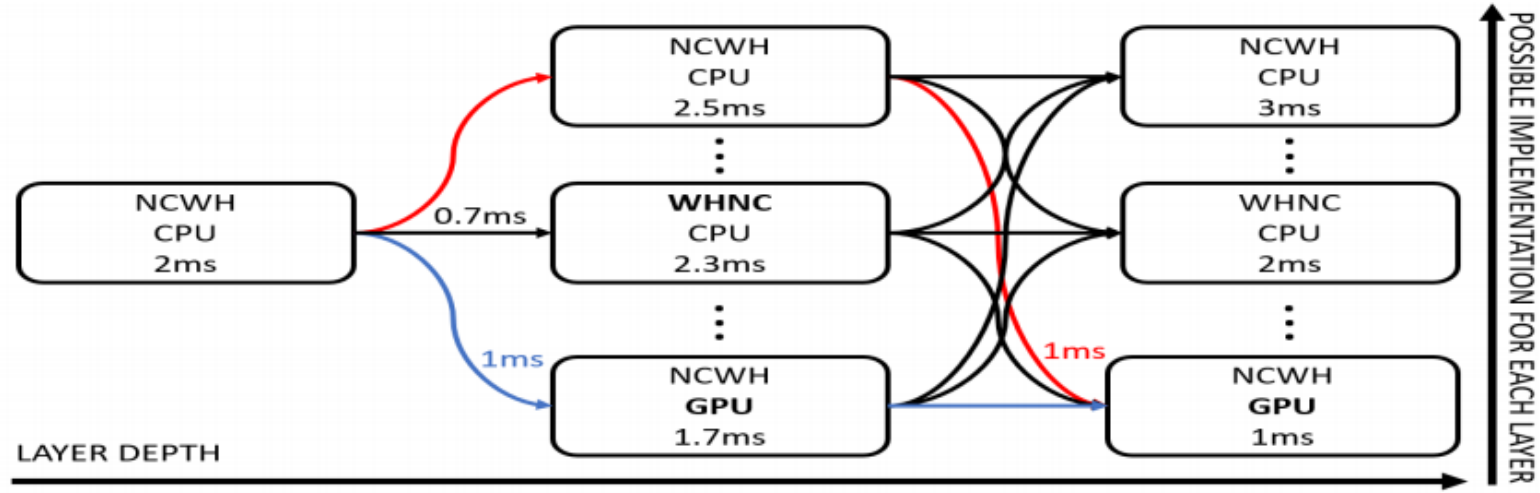
- **Agent (학습의 대상)**
- **Environment (에이전트가 동작하는 환경)**
- **Action/State/Reward**
  - 에이전트가 환경으로 부터 얻은 **보상을 극대화하는 행동을 학습**
  - **Q-learning (Off-policy)**



# Search Engine #1

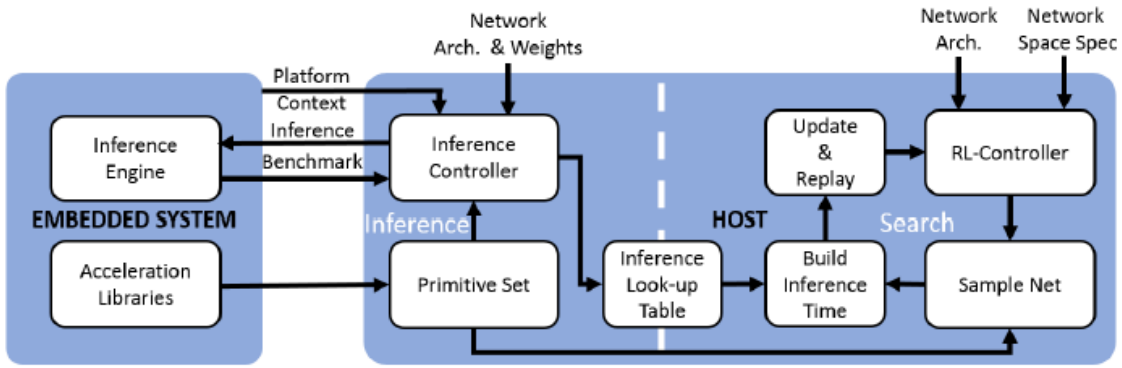
- **State parameter**
- **Action**
  - Selecting best primitive
- **Replay buffer (128)**

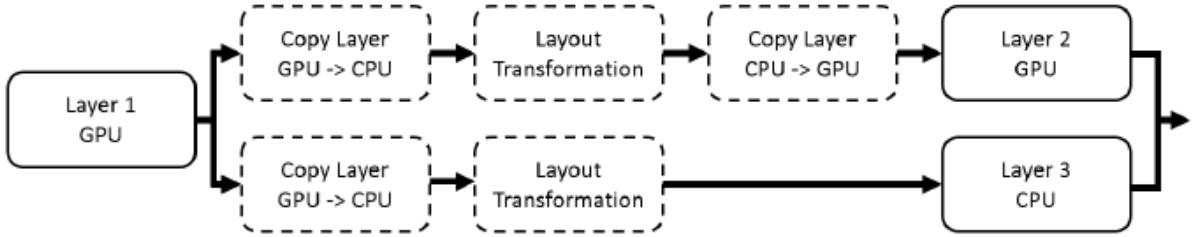
| State Parameters | Definition |
|---|---|
| Layer type | Any layer e.g. convolution, pooling |
| Layer depth | Position of the layer in the network |
| Acceleration Library | Name of the library |
| Algorithm | Routine type |
| Algorithm impl | Sub-routine or lowering method |
| Hardware processor | CPU, GPU, FPGA. |
| BLAS library | Library name |



# Search Engine #2

- **QS-DNN (Q-Based)**
  - Automatically optimize the inference of any DNN
- **Inference look-up table (Inference)**
  - Retrieving All inference measurement
  - Generating the **look-up table**



# Search Engine #3

- **Search space and Conditions**
  - Defined for each network
- **Episode**
  - Each primitive is inferred for 50 images

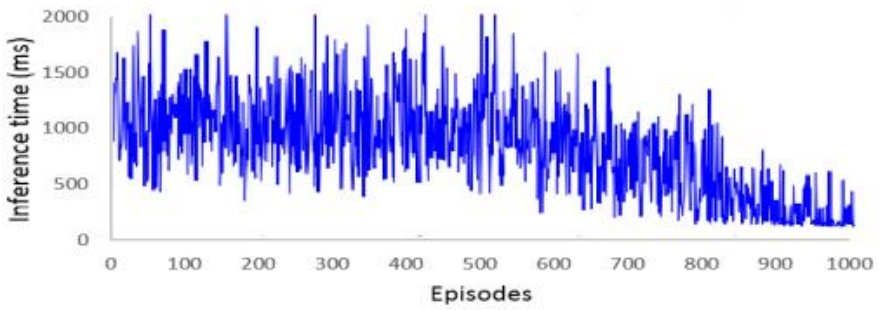



Fig. 4: RL search for 1000 episodes where the 500 first episodes are fully exploration. From there on, $\epsilon$ is decreased by 0.1 towards exploitation after every 50 episodes.

**Algorithm 1** QS-DNN - Search

$\epsilon \leftarrow \epsilon_{new}$
**while** Learned Episodes $<$ Episodes($\epsilon$) **do**
  Reset Path
  **while** Layer $\neq$ End Layer($\epsilon$) **do**
    **if** Generate Random $< \epsilon$ **then**
      Action $\leftarrow$ Q-values(Random)
    **else**
      Action $\leftarrow$ Q-values(Max)
    Layer $\leftarrow$ Next Layer
  Check for Incompatibility
  Compute Inference Time
  Experience Replay & Update (eq. 2)

# Experimental Result

- **Embedded device (Inference target)**
  - Jetson TX-2 (Single precision), Single-thread on Cortex A-57
- **Method**
  - Best Single Library (BSL), Random Search (RS)

| Processor | Method | LeNet-5 | AlexNet | SqueezeNet | MobileNet v1 | MobileNet v2 | GoogleNet | Resnet32 | VGG19 | Mobile-FaceNet | MobileNet v1 SSD |
|---|---|---|---|---|---|---|---|---|---|---|---|
| CPU | Vanilla | x1 | | | | | | | | | |
| | OpenBLAS | 7.69x | 13.45x | 18.55x | 16.32x | 9.54x | 14.79x | 24.95x | 26.79x | 11.38x | 21.14x |
| | NNPACK | 8.41x | 8.82x | 17.53x | 12.61x | 8.30x | 15.63x | 19.76x | 41.36x | 9.66x | 15.81x |
| | ArmCL | 6.27x | 13.38x | 18.50x | 17.31x | 10.58x | 14.37x | 25.01x | 25.45x | 12.61x | 21.58x |
| | RS | 11.31x | 12.84x | 12.94x | 9.12x | 5.26x | 8.92x | 13.71x | 33.14x | 8.43x | 18.72x |
| | **QS-DNN** | **13.02x** | **17.46x** | **21.48x** | **17.89x** | **11.25x** | **17.53x** | **31.06x** | **55.15x** | **13.33x** | **22.34x** |
| | **QS-DNN VS BSL** | **1.55x** | **1.30x** | **1.16x** | **1.03x** | **1.06x** | **1.12x** | **1.24x** | **1.33x** | **1.06x** | **1.04x** |
| GPGPU | cuDNN | 4.06x | 18.88x | 95.14x | 26.39x | 12.23x | 138.61x | 98.36x | 236.96x | 14.43x | 40.03x |
| | RS | 11.89x | 22.33x | 19.11x | 18.51x | 6.74x | 11.19x | 33.10x | 54.60x | 11.38x | 26.73x |
| | **QS-DNN** | **13.02x** | **154.7x** | **95.14x** | **37.59x** | **16.55x** | **166.33x** | **133.07** | **714.46x** | **20.59x** | **48.46x** |
| | **QS-DNN VS BSL** | **3.21x** | **8.19x** | **1.00x** | **1.42x** | **1.35x** | **1.20x** | **1.35x** | **3.02x** | **1.43x** | **1.21x** |

TABLE II: Inference time speedup of CPU- and GPGPU-based implementations respect to Vanilla (dependency-free implementation). Results correspond to most performing libraries employing their fastest primitive for single-thread and 32-bit floating-point operations. QS-DNN VS BSL shows the improvement of the search over the Best Single Library (BSL) and clearly outperforms RS (Random Search) for 1000 episodes.