

SqueezeNet for Leaf Disease Detection

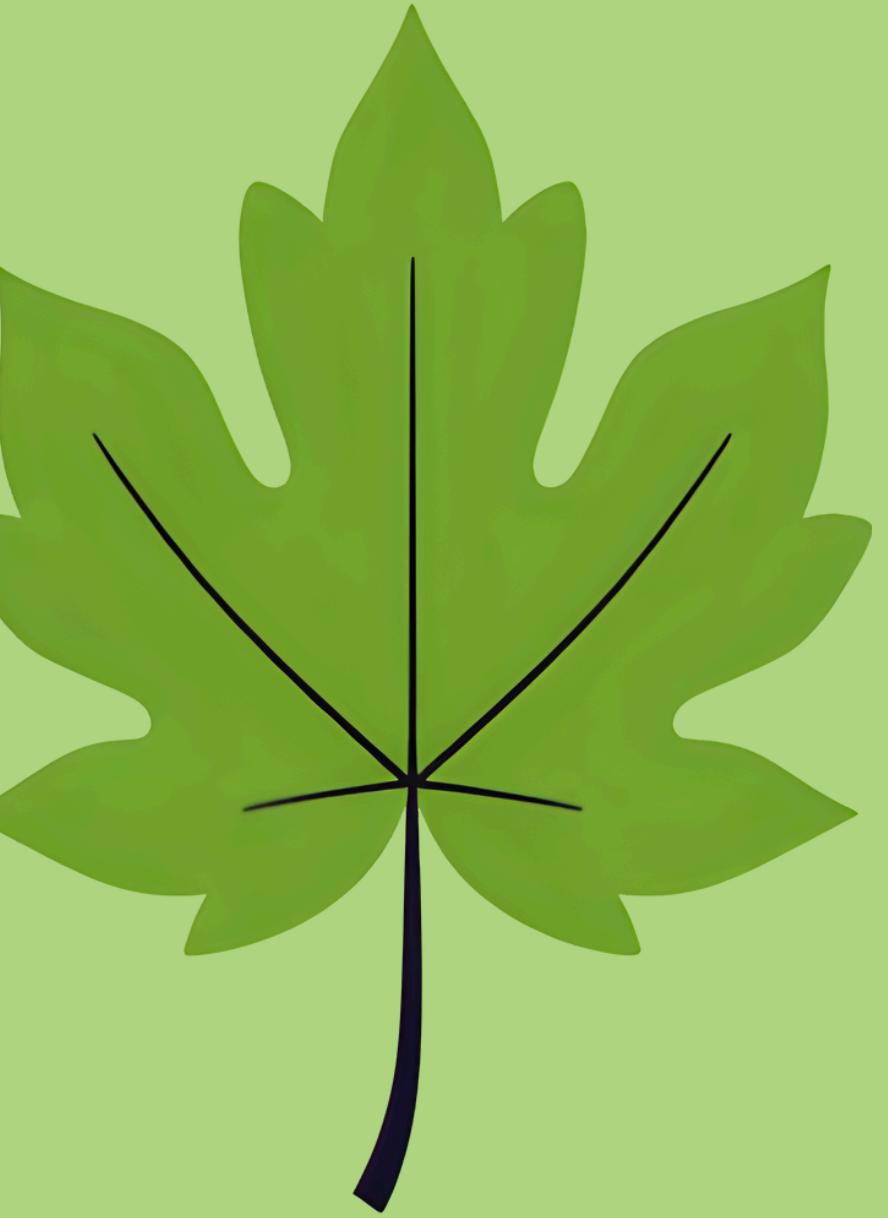
Innovative solutions for agricultural challenges



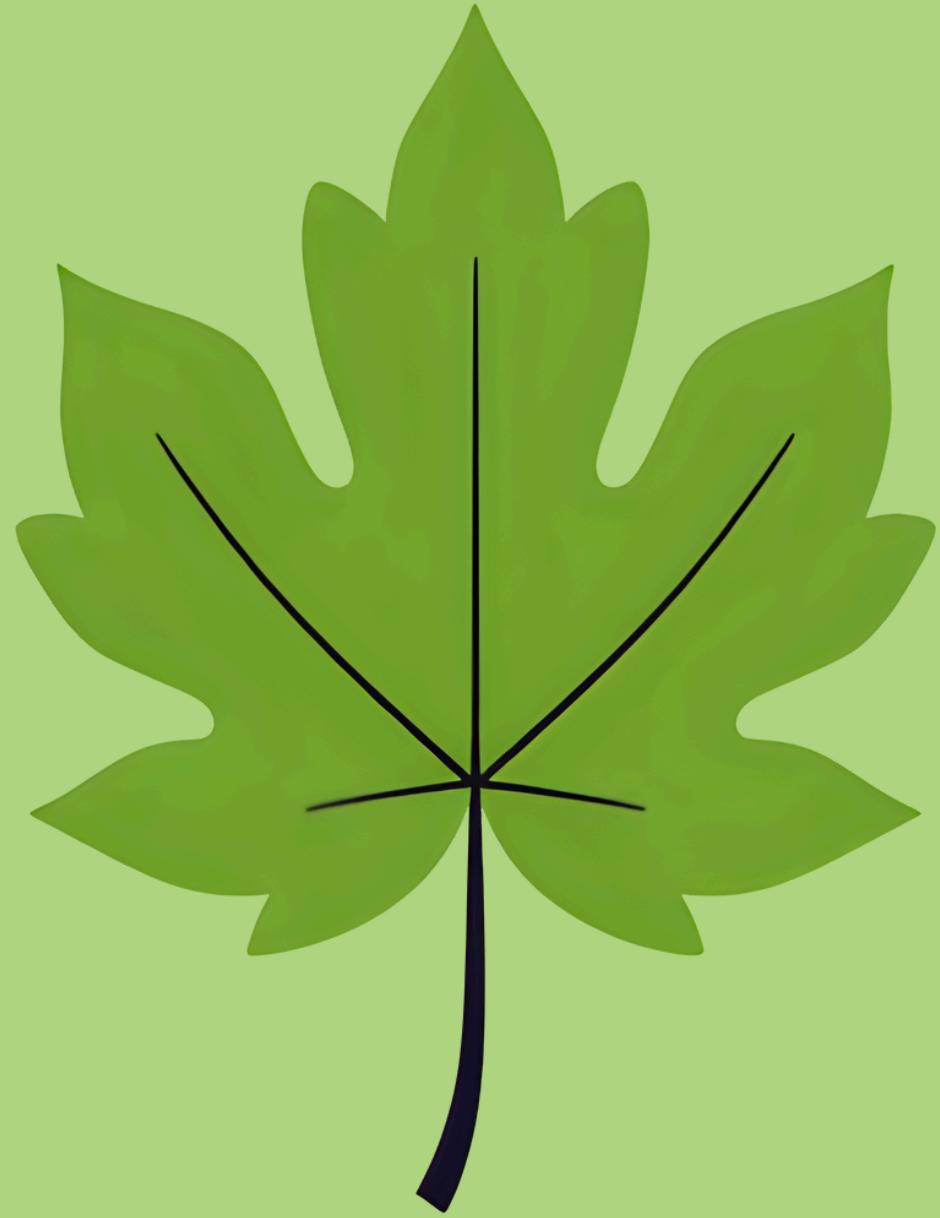


Deleverables Overview

Type of Passing	Modules	Versions			
		CPU-based	CUDA	CUDA with Shared Memory	
Inference	2D Convolution	• Convolution 1x1 • Convolution 3x3 • Convolution 7x7	✓	✓	✓
	Pooling	• Max Pooling • Global Average Pooling	✓	✓	
	Activations	• ReLU • softmax	✓	✓	
	Concat		✓	✓	
Gradient calculation	Gradient w.r.t input	2D Convolution Bias		✓	✓
		Pooling	• Max Pooling • Global Average	✓	✓
		Activations	• ReLU • softmax	✓	✓
	Gradient w.r.t filters	2D Convolution		✓	✓
	Fire Module (Inference and gradient)		✓	✓	



Inference



Convolution

	CPU-based	CUDA	CUDA shared Memory	Pytorch
Input	Images X → 4D np.ndarray / tensor np.ndarray / tensor	Filters W → 4D np.ndarray / tensor Stride S → int Biases B → 1D Padding P → int	Thread Per Block TPB → 16 x 16	
Output		out → 4D np.ndarray / tensor		
Description	2D sliding window algorithm with stride and padding	1 thread per output cell, read directly from global memory, coalesced	Load the input tile into shared memory; reuse the data to reduce global memory access; each block computes the output cells within the tile.	
Time	1x1	2.894	0.00691 (x419)	0.00894 (x324)
	7x7	55.707	0.02662 (x2092)	0.00106 (x52236)

Max Pooling

	CPU-based	CUDA	Pytorch
Input	Images $X \rightarrow$ 4D np.ndarray / tensor Stride $S \rightarrow$ int Padding $P \rightarrow$ int	Thread Per Block TPB $\rightarrow 16 \times 16$	
Output	out \rightarrow 4D np.ndarray / tensor		
Description	<ul style="list-style-type: none">• 2D sliding window algorithm with stride	<ul style="list-style-type: none">• 1 thread per ouput cell, read directly from global memory, coalesced	
Time	9.01628	0.0058 (x1554)	0.00054 (x16522)

Global Average Pooling

	CPU-based	CUDA	Pytorch
Input	Images $X \rightarrow$ 4D np.ndarray / tensor	Thread Per Block TPB $\rightarrow 16 \times 16$	
Output	$out \rightarrow$ 4D np.ndarray / tensor		
Description	<ul style="list-style-type: none">Every cell is divided by $(H \times W)$	<ul style="list-style-type: none">1 thread per output cell, read directly from global memory, coalesced	
Time	0.00024	0.00429 (x0.55)	0.00077 (x0.308)

ReLU

	CPU-based	CUDA	Pytorch
Input	Images $X \rightarrow$ 4D np.ndarray / tensor	Thread Per Block TPB $\rightarrow 16 \times 16$	
Output	$out \rightarrow$ 4D np.ndarray / tensor		
Description	<ul style="list-style-type: none">• Use np.where() to find keep cells' positive values	<ul style="list-style-type: none">• 1 thread per output cell, read directly from global memory, coalesced	
Time	0.05819	0.00584 (x9.957)	0.00095 (x61.142)

Softmax

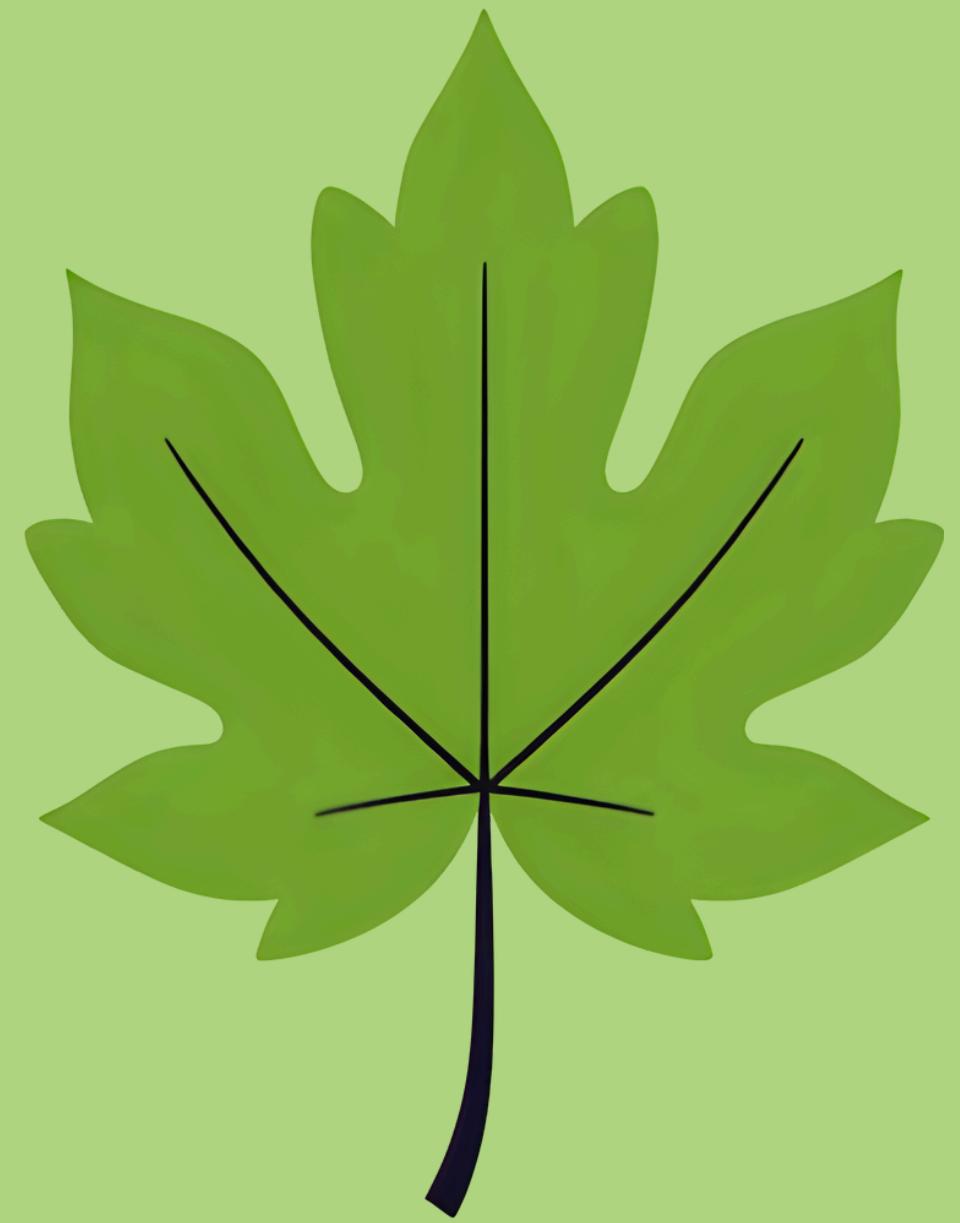
	CPU-based	CUDA	Pytorch
Input	Images X → 4D np.ndarray / tensor	Thread Per Block TPB → 16 x 16	
Output	out → 4D np.ndarray / tensor		
Description	<ul style="list-style-type: none">• Use numpy methods	<ul style="list-style-type: none">• 1 thread per output cell, read directly from global memory, coalesced	
Time	0.00013	0.00365 (x0.036)	0.00094 (x0.141)

Concat

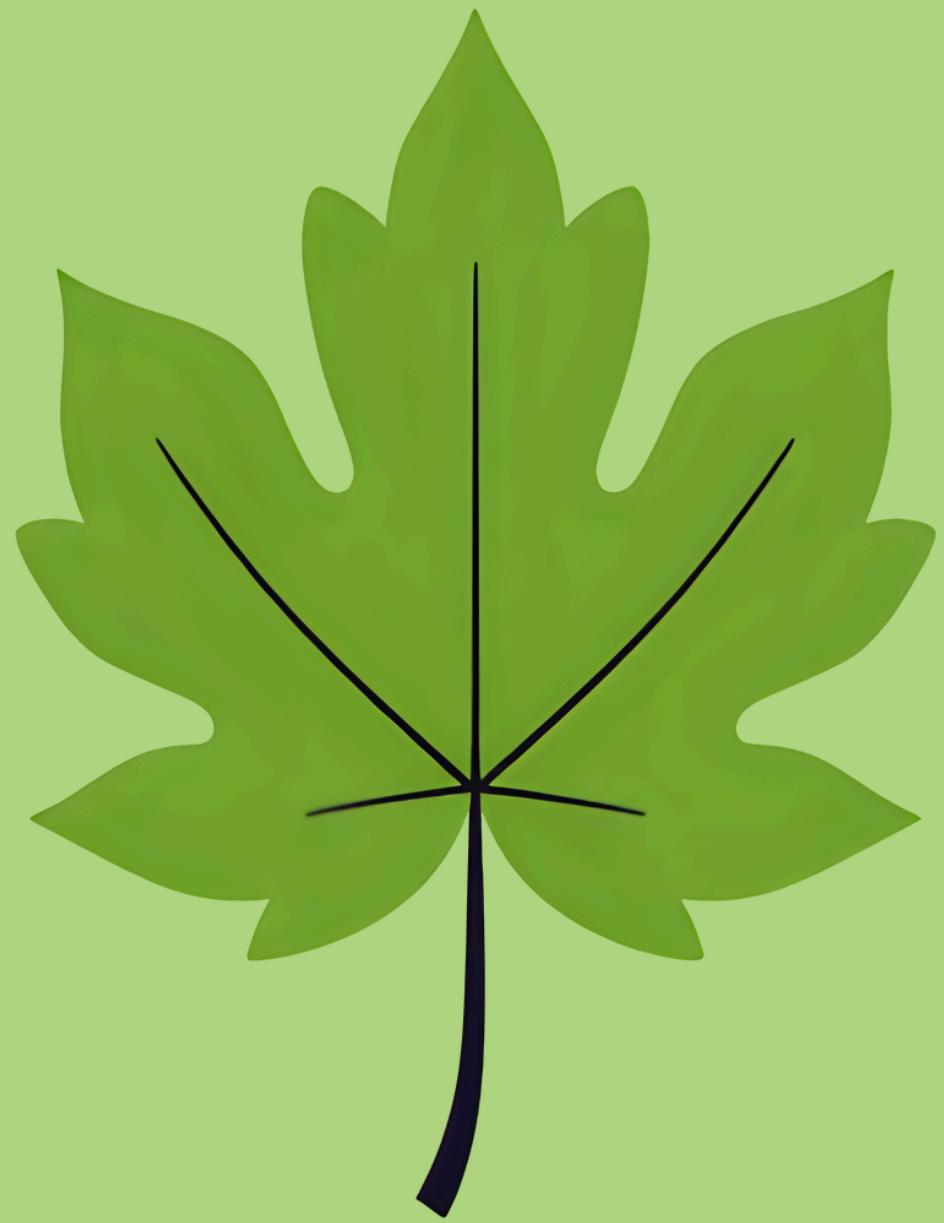
	CPU-based	CUDA	Pytorch
Input	Images X1 → 4D np.ndarray / tensor Images X2 → 4D np.ndarray / tensor	Thread Per Block TPB → 16 × 16	
Output	out → 4D np.ndarray / tensor		
Description	• Use numpy.concatenate() method	• 1 thread per output cell, read directly from global memory, coalesced	
Time	0.00572	0.00459552 (x1.246)	0.00036 (x15.922)

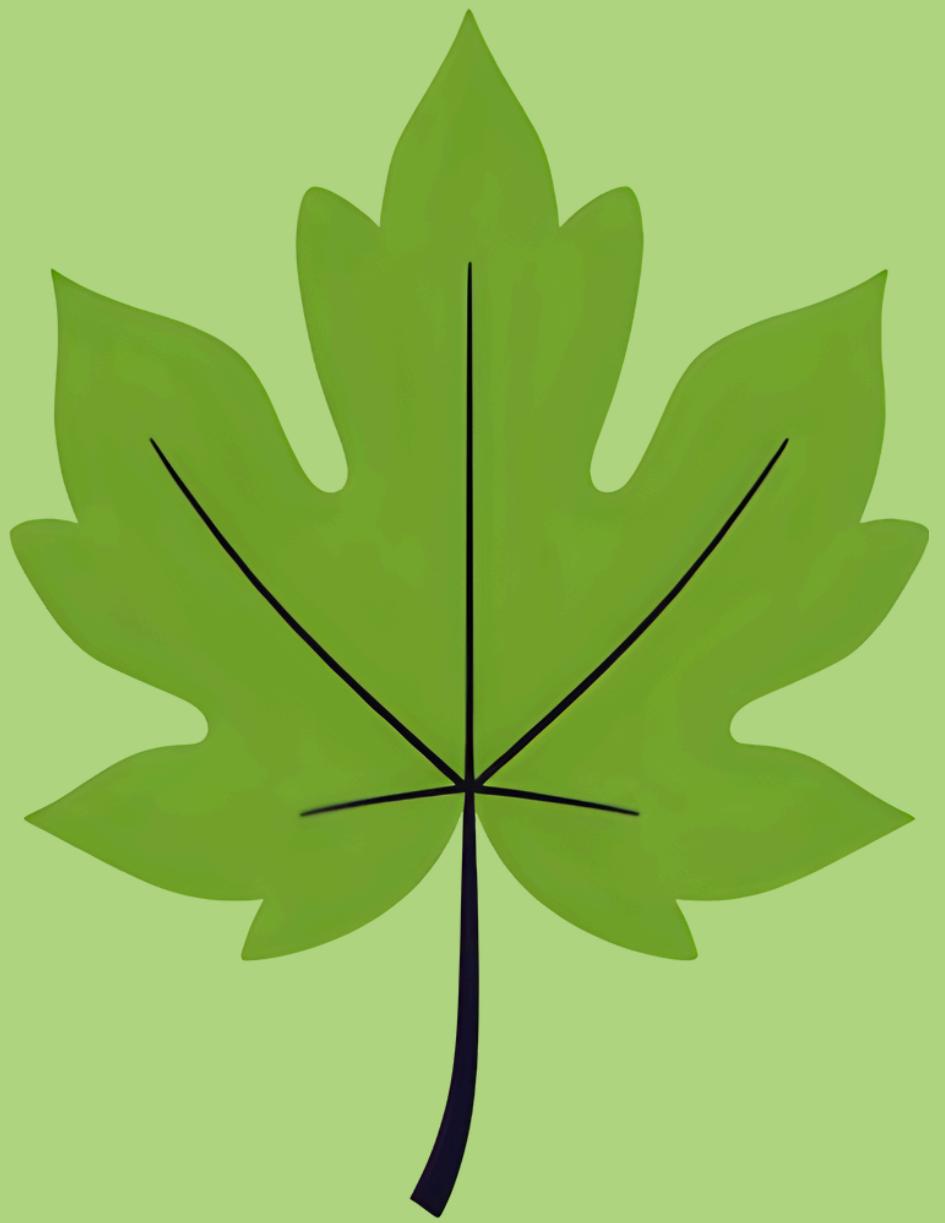
Fire Module

	CPU-based	CUDA	Pytorch
Input	Images X → 4D np.ndarray / tensor		
Output		out → 4D np.ndarray / tensor	
Time	20.0657	0.01832 (x1095.27)	0.00142 (x14085.7)

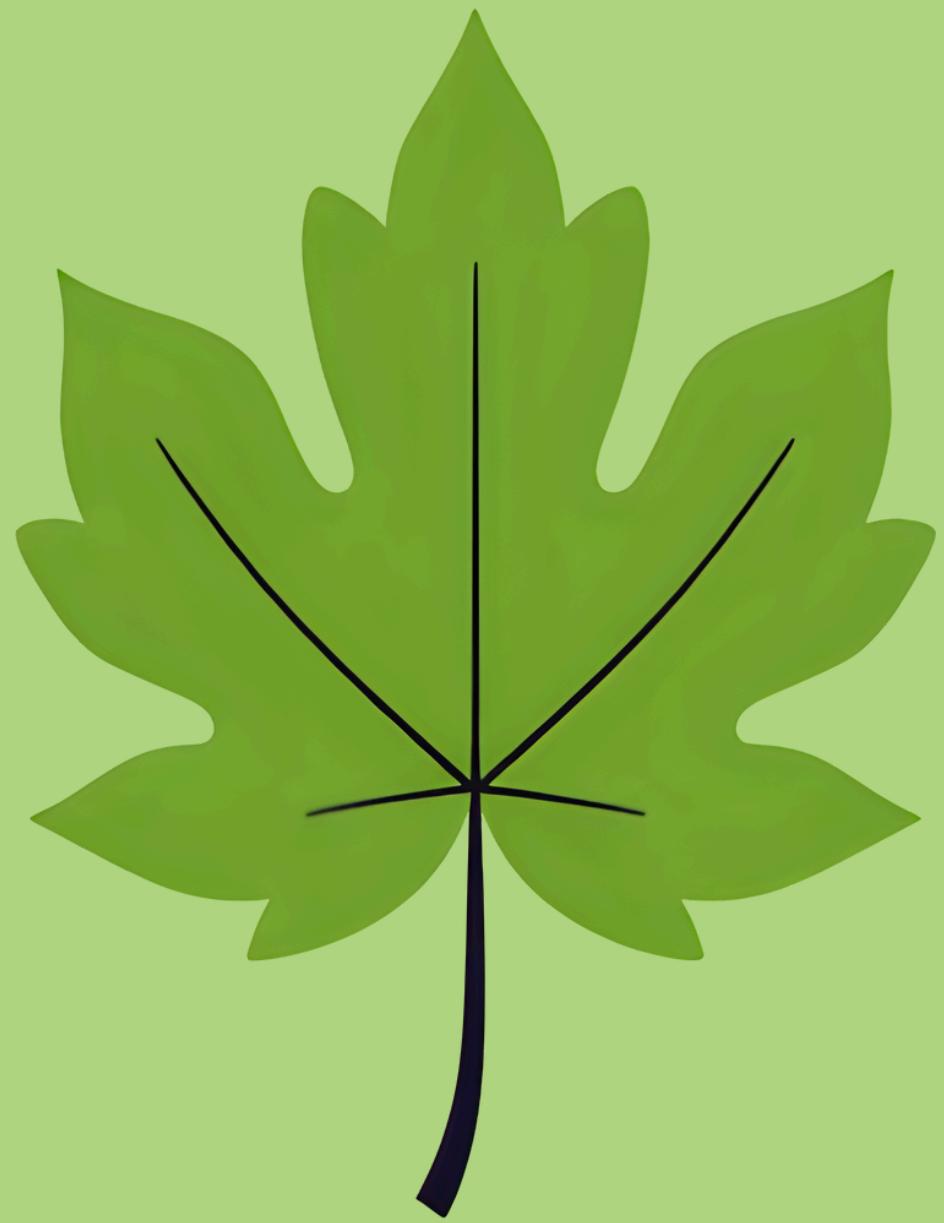


Gradient Calculation





Convolution Gradient w.r.t Input

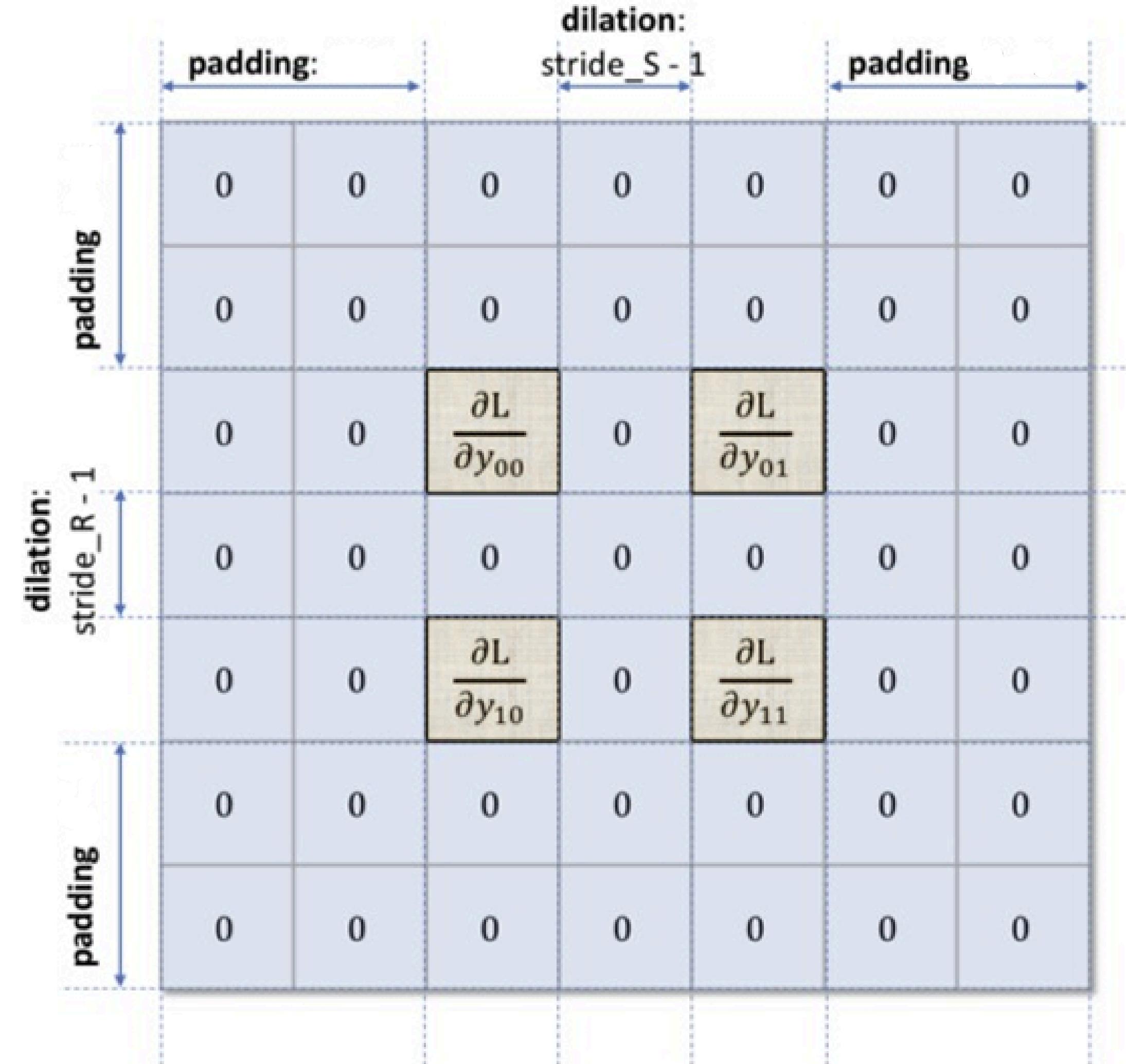


$\frac{\partial L}{\partial y_{00}}$	$\frac{\partial L}{\partial y_{01}}$
$\frac{\partial L}{\partial y_{10}}$	$\frac{\partial L}{\partial y_{11}}$

Output gradients

stride_S = stride_R= 2

Pad and dilate

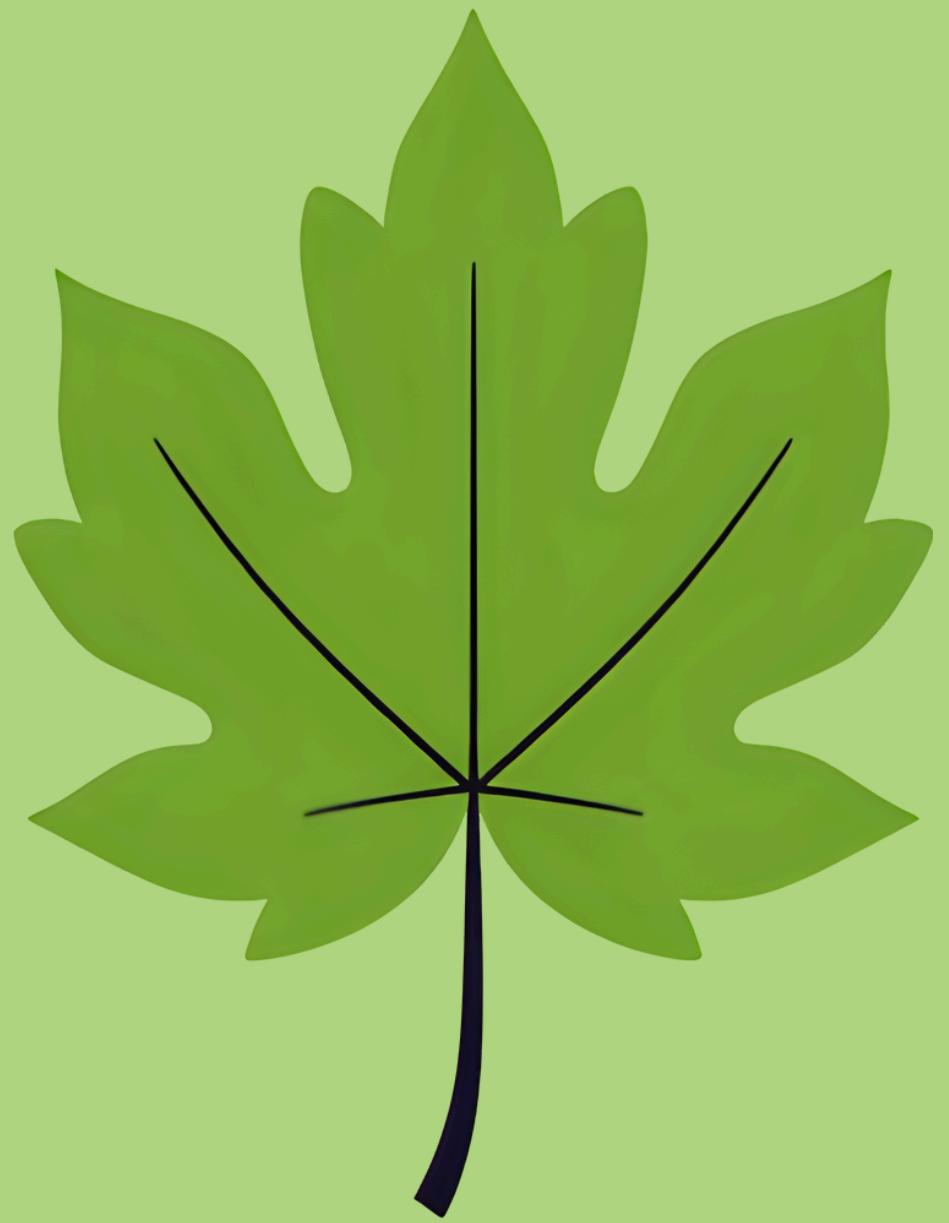


$$\frac{\partial L}{\partial x_{00}} = \frac{\partial L}{\partial y_{00}} f_{00}$$

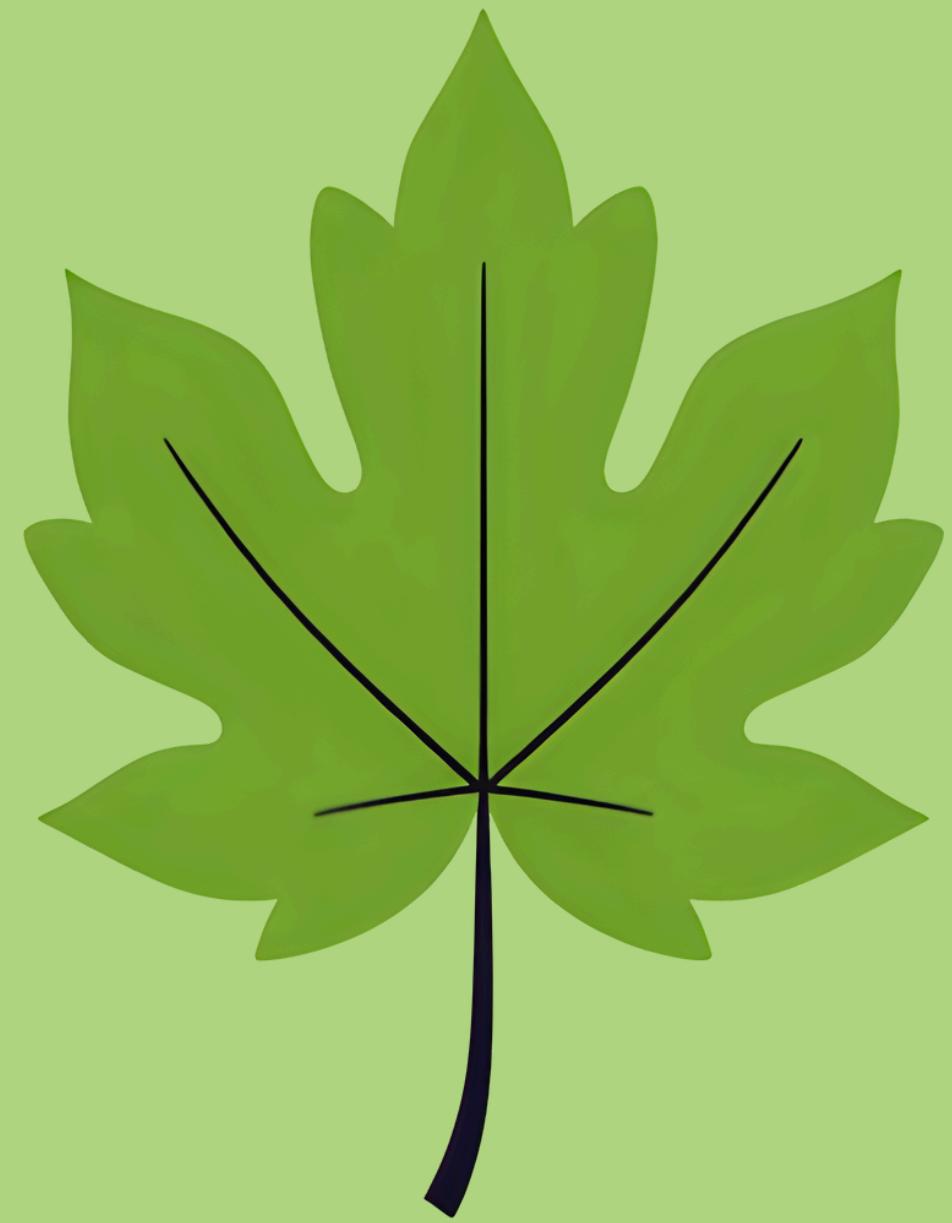
$\frac{\partial L}{\partial x_{00}}$	$\frac{\partial L}{\partial x_{01}}$	$\frac{\partial L}{\partial x_{02}}$	$\frac{\partial L}{\partial x_{03}}$	$\frac{\partial L}{\partial x_{04}}$
$\frac{\partial L}{\partial x_{10}}$	$\frac{\partial L}{\partial x_{11}}$	$\frac{\partial L}{\partial x_{12}}$	$\frac{\partial L}{\partial x_{13}}$	$\frac{\partial L}{\partial x_{14}}$
$\frac{\partial L}{\partial x_{20}}$	$\frac{\partial L}{\partial x_{21}}$	$\frac{\partial L}{\partial x_{22}}$	$\frac{\partial L}{\partial x_{23}}$	$\frac{\partial L}{\partial x_{24}}$
$\frac{\partial L}{\partial x_{30}}$	$\frac{\partial L}{\partial x_{31}}$	$\frac{\partial L}{\partial x_{32}}$	$\frac{\partial L}{\partial x_{33}}$	$\frac{\partial L}{\partial x_{34}}$
$\frac{\partial L}{\partial x_{40}}$	$\frac{\partial L}{\partial x_{41}}$	$\frac{\partial L}{\partial x_{42}}$	$\frac{\partial L}{\partial x_{43}}$	$\frac{\partial L}{\partial x_{44}}$

=

0 * f_{22}	0 * f_{21}	0 * f_{20}	0	0	0	0
0 * f_{12}	0 * f_{11}	0 * f_{10}	0	0	0	0
0 * f_{02}	0 * f_{01}	$\frac{\partial L}{\partial y_{00}} f_{00}$	0	$\frac{\partial L}{\partial y_{01}}$	0	0
0	0	0	0	0	0	0
0	0	$\frac{\partial L}{\partial y_{10}}$	0	$\frac{\partial L}{\partial y_{11}}$	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0



Convolution Gradient w.r.t Filters



$$\frac{\partial L}{\partial f_{00}} = \frac{\partial L}{\partial y_{00}} x_{00} + \frac{\partial L}{\partial y_{01}} x_{02} + \frac{\partial L}{\partial y_{10}} x_{20} + \frac{\partial L}{\partial y_{11}} x_{22}$$

$\frac{\partial L}{\partial f_{00}}$	$\frac{\partial L}{\partial f_{01}}$	$\frac{\partial L}{\partial f_{02}}$
$\frac{\partial L}{\partial f_{10}}$	$\frac{\partial L}{\partial f_{11}}$	$\frac{\partial L}{\partial f_{12}}$
$\frac{\partial L}{\partial f_{20}}$	$\frac{\partial L}{\partial f_{21}}$	$\frac{\partial L}{\partial f_{22}}$

=

$\frac{\partial L}{\partial y_{00}} x_{00}$	$0 * x_{01}$	$\frac{\partial L}{\partial y_{01}} x_{02}$	x_{03}	x_{04}
$0 * x_{10}$	$0 * x_{11}$	$0 * x_{12}$	x_{13}	x_{14}
$\frac{\partial L}{\partial y_{10}} x_{20}$	$0 * x_{21}$	$\frac{\partial L}{\partial y_{11}} x_{22}$	x_{23}	x_{24}
x_{30}	x_{31}	x_{32}	x_{33}	x_{34}
x_{40}	x_{41}	x_{42}	x_{43}	x_{44}

Gradient Convolution

	CPU-based	CUDA
Input	Images X → 4D np.ndarray / tensor Gradient pre-layer Output dY → 4D np.ndarray / tensor	Filters W → 4D np.ndarray / tensor Stride S → int Padding P → int
Output	dX → 4D np.ndarray / tensor	dW → 4D np.ndarray / tensor dB → 4D np.ndarray / tensor
Description	2D sliding window algorithm with stride	1 thread per output cell, read directly from global memory, coalesced
Time	87.8563	0.11775 (x746.131)

Gradient ReLU

	CPU-based	CUDA
Input	Images X → 4D np.ndarray / tensor Gradient pre-layer Output dY → 4D np.ndarray / tensor	Thread Per Block TPB → 16 × 16
Output	dX → 4D np.ndarray / tensor	
Description	2D sliding window algorithm with stride	1 thread per ouput cell, read directly from global memory, coalesced
	Keep gradient cell if it is positive, else assign zero	
Time	0.00358	0.00520 (x0.688)

Gradient Max Pooling

	CPU-based	CUDA
Input	<p>Images $X \rightarrow$ 4D np.ndarray / tensor</p> <p>Gradient pre-layer Output $dY \rightarrow$ 4D np.ndarray / tensor</p>	<p>Thread Per Block TPB $\rightarrow 16 \times 16$</p>
Output	<p>$dX \rightarrow$ 4D np.ndarray / tensor</p>	
Description	<p>2D sliding window algorithm with stride</p>	<p>1 thread per ouput cell, read directly from global memory, coalesced, using atomic add</p>
	Find out all the positions of max value, pass gradient to these indice	
Time	<p>11.4671</p>	<p>0.00669 (x1714.309)</p>

Gradient Global Average Pooling

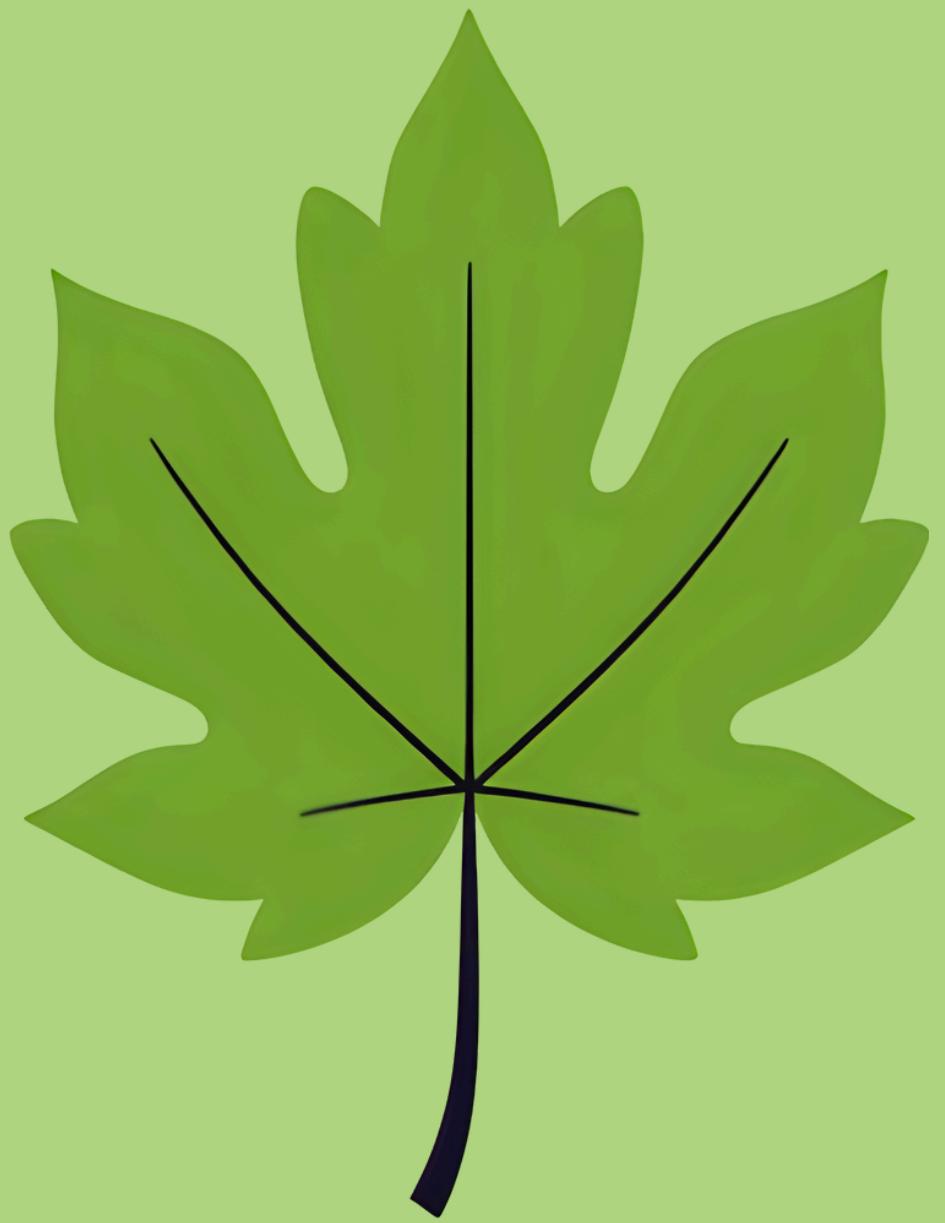
	CPU-based	CUDA
Input	Gradient pre-layer Output $dY \rightarrow$ 4D np.ndarray / tensor	Thread Per Block TPB $\rightarrow 16 \times 16$
Output	$dX \rightarrow$ 4D np.ndarray / tensor	
Description	Use numpy.average() on all dimensions	1 thread per output cell, read directly from global memory, coalesced
Time	0.00054	0.00401 (x0.134)

Gradient Softmax

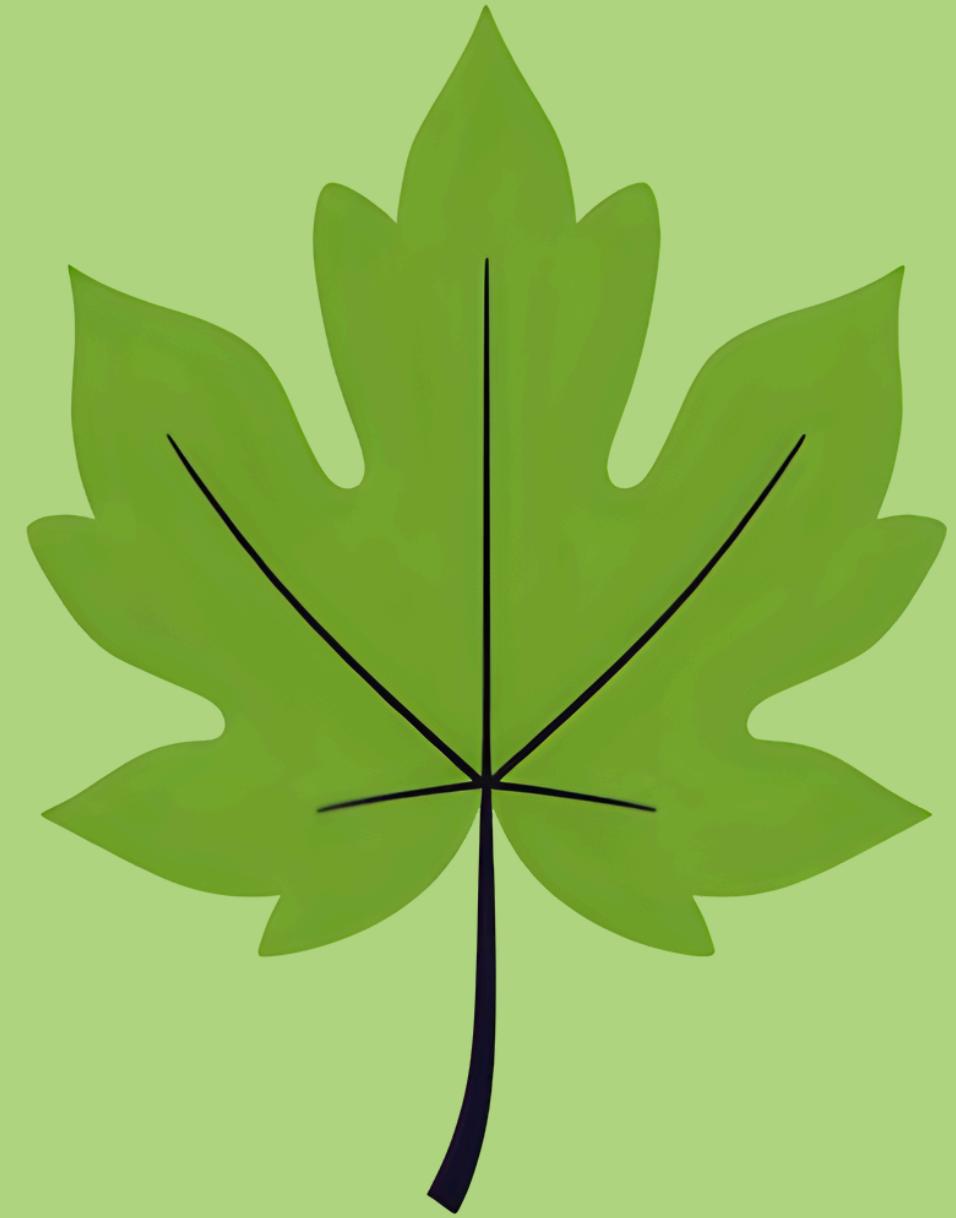
	CPU-based	CUDA
Input	probs (B,N) → 2D np.ndarray / tensor y_onehot → 2D np.ndarray / tensor	Thread Per Block TPB → 16 x 16
Output	dLogits → 4D np.ndarray / tensor	
Description	cross-entropy: $dL/dz = (p - y)/B$.	1 thread per ouput cell, read directly from global memory, coalesced
Time	0.00014	0.00401 (x0.035)

Gradient Fire Module

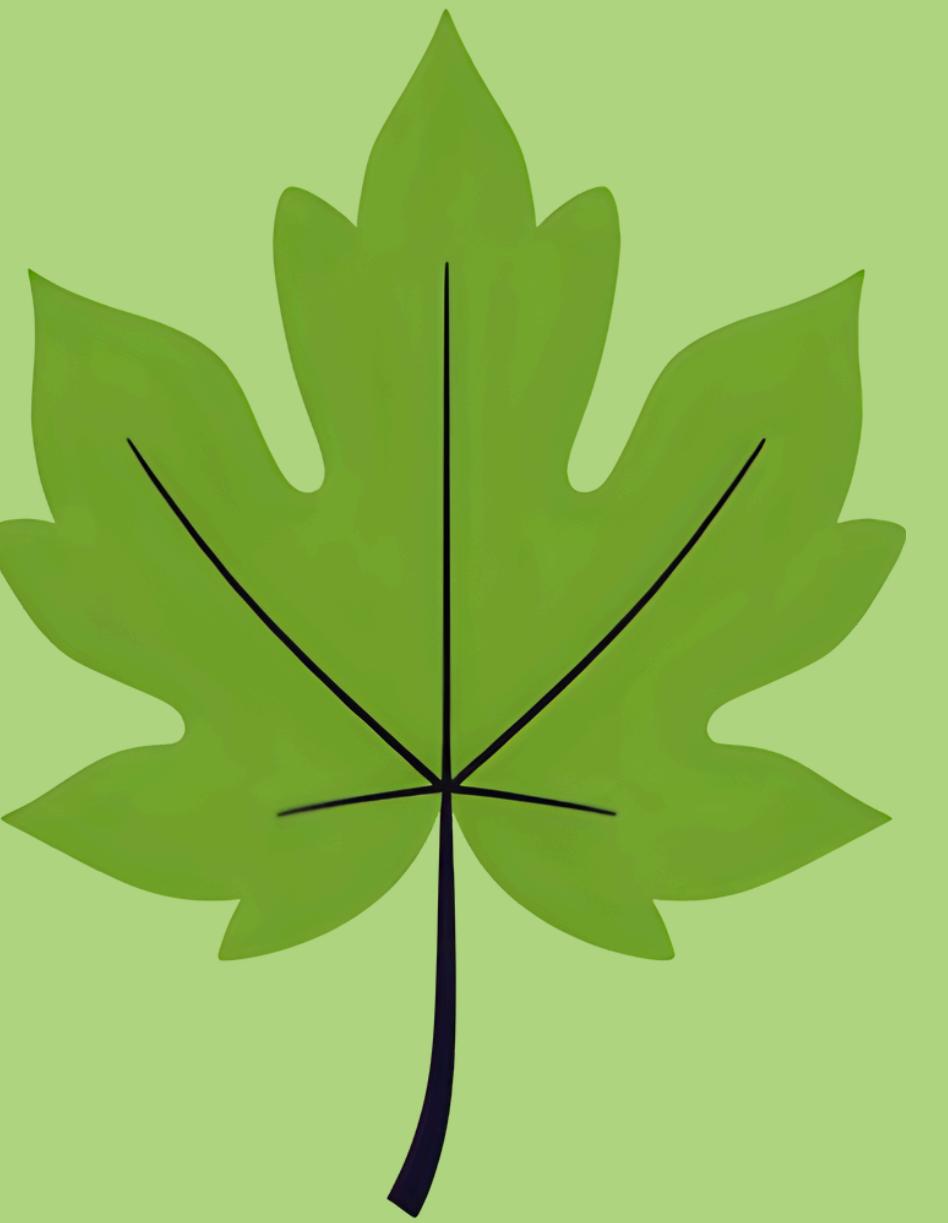
	CPU-based	CUDA
Input	<p>Images (X_prev) → 4D np.ndarray / 4D tensor, Input activations from the previous layer.</p> <p>Squeeze Weights (W_sq1x1) → 4D np.ndarray / 4D tensor Parameters of the squeeze (1×1 conv).</p> <p>Squeeze Output (X_sq) → 4D np.ndarray / 4D tensor Cached outputs needed for gradient masking.</p> <p>Expand 1×1 Weights (W_ex1x1) → 4D np.ndarray / 4D tensor</p> <p>Expand 3×3 Weights (W_ex3x3) → 4D np.ndarray / 4D tensor</p> <p>Concat Output (X_concat) → 4D np.ndarray / 4D tensor Forward concat of expand1×1 + expand3×3.</p> <p>Gradient w.r.t Fire Output (dY) → 4D np.ndarray / 4D tensor Gradient coming from the next layer.</p>	<p>Squeeze Bias (b_sq1x1) → 1D np.ndarray / 1D tensor</p> <p>ReLU(Squeeze) Output (X_sq_relu) → 4D np.ndarray / 4D tensor</p> <p>Expand 1×1 Bias (b_ex1x1) → 1D np.ndarray / 1D tensor</p> <p>Expand 3×3 Bias (b_ex3x3) → 1D np.ndarray / 1D tensor</p>
Output	<p>dX_prev → 4D np.ndarray / 4D tensor</p> <p>dW_sq1x1 → 4D np.ndarray / 4D tensor</p> <p>dW_ex1 → 4D np.ndarray / 4D tensor</p> <p>dW_ex3 → 4D np.ndarray / 4D tensor</p>	<p>db_sq1x1 → 1D np.ndarray / 1D tensor</p> <p>db_ex1 → 1D np.ndarray / 1D tensor</p> <p>db_ex3 → 1D np.ndarray / 1D tensor</p>
Time	<p>31.0295</p>	<p>0.157624 (x196.858)</p>



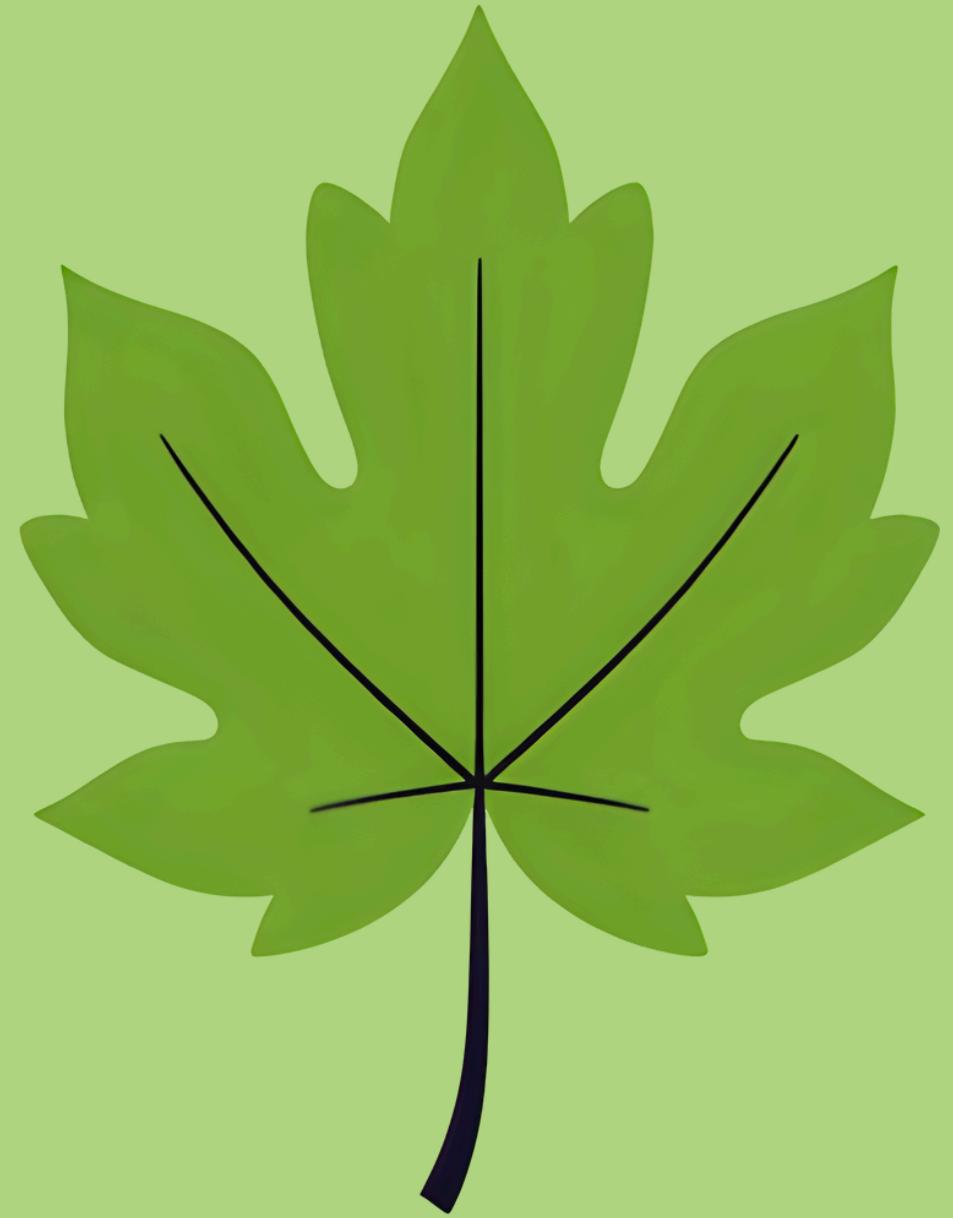
**Forward
Backward**



	CPU-based	CUDA
Input	<p>Images → 4D np.ndarray / 4D tensor</p> <p>Combine all module to model SqueezeNet (both forward and backward)</p>	
Output forward		A dictionary of all layer in SqueezeNet
Output backward		A gradient dictionary of all weight and bias in SqueezeNet
Time forward	227.427	0.195301 (x1164.495)
Time backward	350.383	1.32089 (x265.2628)
Time train in 1 iteration	577.814	1.51619 (x381.097)



Training



Optimizer

Main Idea

- Use the Adam optimizer to adjust the learning rate, thereby updating gradients more efficiently.
- Since this is a custom optimization problem, instead of creating an update function like PyTorch's `optimizer.step`, we update directly during gradient computation to avoid slowing down the program

$$\begin{aligned} t &\leftarrow t + 1 \\ g_t &\leftarrow \nabla_{\theta} f_t(\theta_{t-1}) \text{ (Get gradients w.r.t. stochastic objective at timestep } t) \\ m_t &\leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t \text{ (Update biased first moment estimate)} \\ v_t &\leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2 \text{ (Update biased second raw moment estimate)} \\ \hat{m}_t &\leftarrow m_t / (1 - \beta_1^t) \text{ (Compute bias-corrected first moment estimate)} \\ \hat{v}_t &\leftarrow v_t / (1 - \beta_2^t) \text{ (Compute bias-corrected second raw moment estimate)} \\ \theta_t &\leftarrow \theta_{t-1} - \alpha \cdot \hat{m}_t / (\sqrt{\hat{v}_t} + \epsilon) \text{ (Update parameters)} \end{aligned}$$

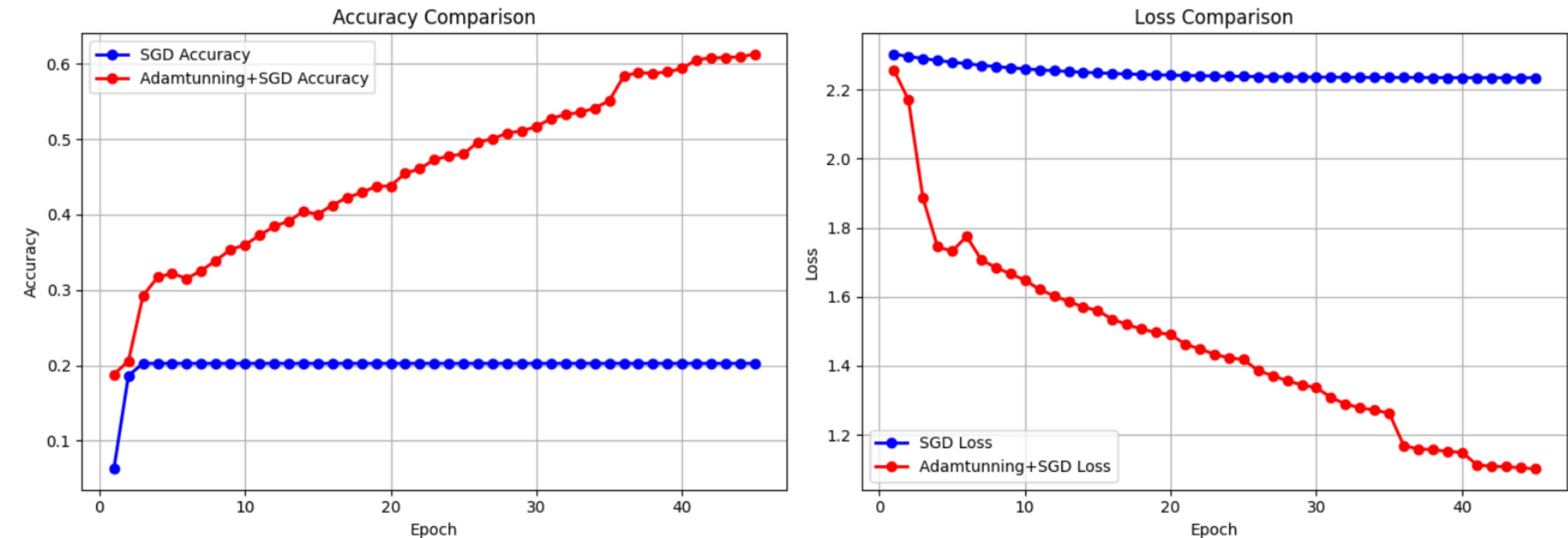
	CUDA+AdamTuning+SGD	Pytorch+SGD
Input	Number of batch : 550 Batch size: 32x3x224x224	
Precision (45 epochs)	61.3%	20.25%

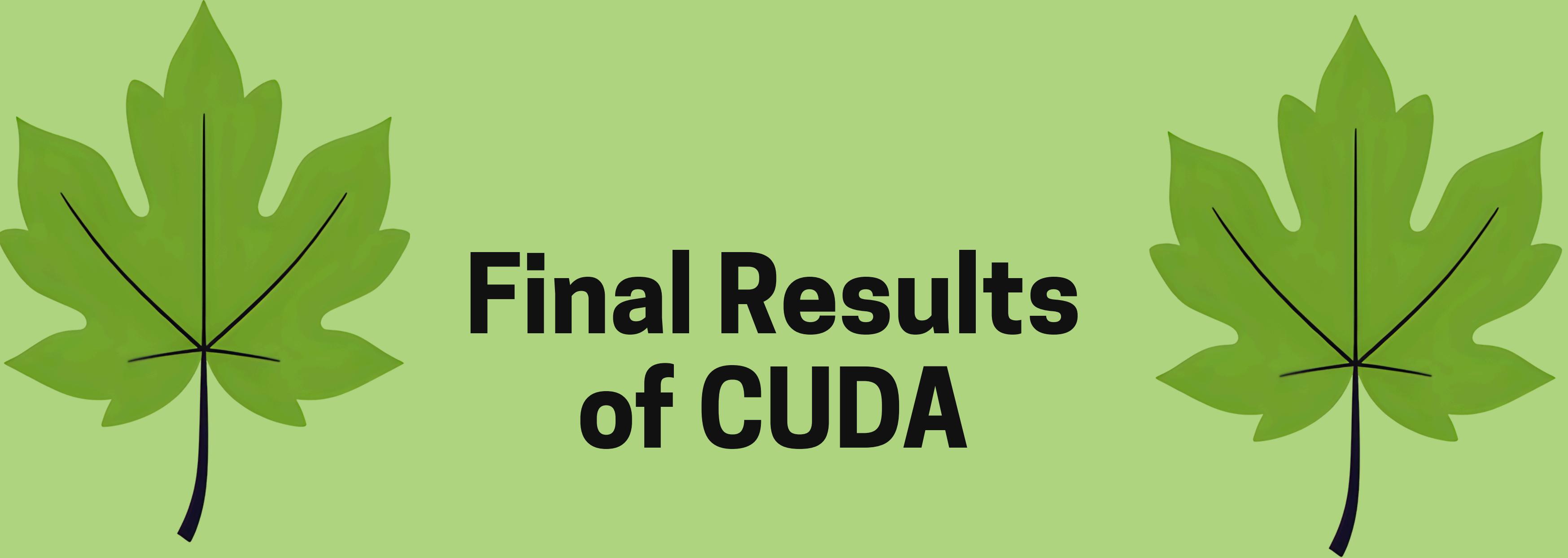
Cuda Memory Management

Main Idea

- The `cuda.device(A)` operation transfers array A to CUDA memory. Unlike regular arrays or PyTorch tensors on CPU, CUDA memory is not automatically released, requiring manual deallocation to free up space for model training.
- Our team uses `cuda.get_current_device().reset()` to release CUDA memory. However, before resetting, you must move the model's weights and optimizer state back to the CPU, as this function clears all data on the CUDA device.

Cuda Squeenet Training





Final Results of CUDA



Forward

- ×1164 speedup
- 90% accuracy,
- 100% similar with Pytorch
- 0.017s inferent with image(3x224x224)



Backward

- GPU pass 256x speedup
- 100% similar with Pytorch
- Implement Adam optimizer



Training

- 381x speedup
- 0% → 63% accuracy in 45 epochs
- Implement Adam, SGD optimizer

