Deep Learning from First Principles

Lesson 9 - CNN Improvements

Andrea Giardina
contact@andreagiardina.com
https://www.linkedin.com/in/agiardina

From naive Conv/Pool to batched operations

Motivation

- Our first Conv2d and MaxPool2d implementations were *naive but clear*: explicit Python loops over the batch.
- This is fine to understand the mechanics, but becomes inefficient as soon as we increase the batch size or add more layers.

Goal of this block

- Keep the same high-level API:

$$x \in \mathbb{R}^{N \times C \times H \times W} \;\Rightarrow\; y \in \mathbb{R}^{N \times C' \times H' \times W'}.$$

- Remove Python loops over the batch dimension.
- Use the im2col / col2im trick on the whole batch: turn convolution and max pooling into large matrix operations that NumPy (BLAS) can execute efficiently.

Today: we will refactor Conv2d and MaxPool2d to their batched versions, reusing exactly the same ideas we already used for the single-image case.

MaxPool2d: naive implementation (single image)

Goal of MaxPool2d:

- Reduce the spatial resolution (H, W)
- Keep, for each local patch, only the maximum value
- Operate independently on each channel

Naive implementation (single image at a time):

- Input batch: $x \in \mathbb{R}^{N \times C \times H \times W}$
- For each sample n in the batch: extract $x_n \in \mathbb{R}^{C \times H \times W}$
- For each channel c:
  - Apply `im2col` on $x_{n,c}$ (single-image version)
  - Take the maximum over each column

Problem: we have explicit loops (`for n`, `for c`) $\Rightarrow$ inefficient for large batches.

From single image to batch: key idea

Question: can we apply the `im2col` trick directly on the whole batch?

Single-image version:

$$x_n \in \mathbb{R}^{C \times H \times W} \;\Rightarrow\; X_{\text{col}}^{(n)} \in \mathbb{R}^{(C k_H k_W) \times L}$$

Batched version:

$$x \in \mathbb{R}^{N \times C \times H \times W} \;\Rightarrow\; X_{\text{col}} \in \mathbb{R}^{N \times (C k_H k_W) \times L}$$

where:

- N = batch size
- k_H, k_W = kernel height and width
- $L = H_{\text{out}} \cdot W_{\text{out}}$ = number of spatial positions

Result: all samples in the batch are transformed at once, without explicit Python loops over N.

MaxPool2d batched: forward (concept)

Forward pass with batched `im2col`:

1. Apply batched `im2col`:

$$X_{\text{col}} \in \mathbb{R}^{N \times (Ck_H k_W) \times L}$$

2. Reshape to separate channels and kernel positions:

$$X_{\text{col}} \to \tilde{X} \in \mathbb{R}^{N \times C \times (k_H k_W) \times L}$$

3. Take the maximum along the kernel-position axis:

$$Y = \max_{\text{pos} \in \{1, \ldots, k_H k_W\}} \tilde{X} \quad \Rightarrow \quad Y \in \mathbb{R}^{N \times C \times L}$$

4. Reshape Y back to $Y \in \mathbb{R}^{N \times C \times H_{\text{out}} \times W_{\text{out}}}$.

Important: besides Y, we also store the indices of the maxima (for each (n, c, ℓ)), so that we can route gradients correctly in the backward pass.

MaxPool2d batched: forward (pseudo-code)

```
class MaxPool2d:
    def forward(self, x):
        # x: (N, C, H, W)
        N, C, H, W = x.shape
        kH, kW = self.kernel_size
        p = self.padding
        s = self.stride

        # pad input if needed
        if p > 0:
            x_padded = np.pad(
                x, ((0, 0), (0, 0), (p, p), (p, p)),
                mode="constant"
            )
        else:
            x_padded = x

        # batched im2col works on already padded input
        X_col = im2col_batch(x_padded, kH, kW, stride=s)
        # X_col: (N, C * kH * kW, L)

        N2, CK, L = X_col.shape
        assert N2 == N
        X_reshaped = X_col.reshape(N, C, kH * kW, L)
```

MaxPool2d batched: backward (vectorized idea)

In the backward pass we receive:

$$\frac{\partial \mathcal{L}}{\partial Y} \in \mathbb{R}^{N \times C \times H_{\text{out}} \times W_{\text{out}}},$$

which we reshape to

$$dY_{\text{flat}} \in \mathbb{R}^{N \times C \times L}, \quad L = H_{\text{out}} \cdot W_{\text{out}}.$$

Goal: we want to place each gradient value $dY_{\text{flat}}[n, c, \ell]$ into the correct position in the column representation $dX_{\text{col}}[n, c, \text{pos}, \ell]$ for all (n, c, ℓ) at once, without explicit Python loops.

Vectorized indexing strategy:

1. Create index grids:

   $$n\_idx = 0, \ldots, N-1, \quad c\_idx = 0, \ldots, C-1, \quad \ell\_idx = 0, \ldots, L-1$$

   broadcasted to shape (N, C, L).
2. Use the stored $\text{max\_idx} \in \mathbb{R}^{N \times C \times L}$ as the third index (position inside the kernel).
3. One shot assignment:

   $$dX_{\text{cols\_reshaped}}[n\_idx, c\_idx, \text{max\_idx}, \ell\_idx] = dY_{\text{flat}}[n\_idx, c\_idx, \ell\_idx].$$

   Conceptually this corresponds to a "$+=$" update on the original image, but the accumulation actually happens later when we apply `col2im`.
4. Finally, apply batched `col2im` to go from dX_{cols} back to $\frac{\partial \mathcal{L}}{\partial X} \in \mathbb{R}^{N \times C \times H \times W}$.

Conceptually, this is the same as the triple loop over (n, c, ℓ), but NumPy does all iterations internally in C.

MaxPool2d backward: local view for one patch

To better understand the backward pass, fix a single triplet (n, c, ℓ):

- In the forward pass, after `im2col` and reshaping, we had

$$\tilde{X}[n, c, :, \ell] \in \mathbb{R}^{k_H k_W}$$

  which is the flattened $k_H \times k_W$ patch.
- We computed

$$y[n, c, \ell] = \max_{j=0,\dots,k_H k_W - 1} \tilde{X}[n, c, j, \ell]$$

  and stored

$$\mathrm{pos} = \mathrm{max\_idx}[n, c, \ell] \in \{0, \dots, k_H k_W - 1\}.$$

- In the backward pass we receive a scalar gradient

$$g = \frac{\partial \mathcal{L}}{\partial y[n, c, \ell]}.$$

  The gradient w.r.t. the vector $\tilde{X}[n, c, :, \ell]$ is:

$$\frac{\partial \mathcal{L}}{\partial \tilde{X}[n, c, j, \ell]} = \begin{cases} g & \text{if } j = \mathrm{pos} \\ 0 & \text{otherwise.} \end{cases}$$

Code hint: for a naive implementation, you can literally write `for n, for c, for l`, compute `pos`, and assign `dX_cols_reshaped[n, c, pos, l] += dY_flat[n, c, l]`.

Exercise: implementing batched MaxPool2d

Exercise:

1. Implement a function `im2col_batch(x, kH, kW, stride)` that maps $x \in \mathbb{R}^{N\times C\times H\times W}$ to $X_{\text{col}} \in \mathbb{R}^{N\times(Ck_H k_W)\times L}$. Assume that x is already padded if needed.
2. Rewrite the `forward` method of `MaxPool2d` using `im2col_batch`, without loops over the batch.
3. Implement the vectorized `backward` method using the stored `max_idx` and a `col2im_batch` function.

Hint: first focus on getting all the **shapes** consistent, then check numerical correctness by comparing with the naive implementation.

Solution: batched im2col

```
import numpy as np


def im2col_batch(x, kH, kW, stride=1):
    N, C, H, W = x.shape
    H_out = (H - kH) // stride + 1
    W_out = (W - kW) // stride + 1
    L = H_out * W_out

    X_col = np.empty((N, C * kH * kW, L), dtype=x.dtype)
    col = 0
    for i in range(0, H - kH + 1, stride):
        for j in range(0, W - kW + 1, stride):
            patch = x[:, :, i:i + kH, j:j + kW]
            X_col[:, :, col] = patch.reshape(N, -1)
            col += 1
    return X_col
```

Solution: batched col2im

```
def col2im_batch(X_col, C, H, W, kH, kW, stride=1):
    N = X_col.shape[0]
    x = np.zeros((N, C, H, W), dtype=X_col.dtype)

    col = 0
    for i in range(0, H - kH + 1, stride):
        for j in range(0, W - kW + 1, stride):
            patch = X_col[:, :, col].reshape(N, C, kH, kW)
            x[:, :, i:i + kH, j:j + kW] += patch
            col += 1
    return x
```

Solution: MaxPool2d (init + forward)

```python
class MaxPool2d:
    def __init__(self, kernel_size, stride=None, padding=0):
        if isinstance(kernel_size, int):
            kernel_size = (kernel_size, kernel_size)
        self.kernel_size = kernel_size
        self.stride = stride if stride is not None else kernel_size
        self.padding = padding
        self.cache = None

    def forward(self, x):
        N, C, H, W = x.shape
        kH, kW = self.kernel_size
        p = self.padding
        s = self.stride

        if p > 0:
            x = np.pad(x, ((0, 0), (0, 0), (p, p), (p, p)),
                       mode="constant")

        _, _, H_pad, W_pad = x.shape
        H_out = (H_pad - kH) // s + 1
        W_out = (W_pad - kW) // s + 1
        L = H_out * W_out
```

Solution: MaxPool2d (forward + cache)

```
        X_col = im2col_batch(x, kH, kW, stride=s)
        X_reshaped = X_col.reshape(N, C, kH * kW, L)

        max_idx = X_reshaped.argmax(axis=2)
        out = X_reshaped.max(axis=2).reshape(N, C, H_out, W_out)

        self.cache = {
            "x_padded_shape": x.shape,
            "max_idx": max_idx,
        }

        return out

    def backward(self, dY):
        kH, kW = self.kernel_size
        p = self.padding
        s = self.stride

        N, C, H_out, W_out = dY.shape
        L = H_out * W_out

        x_padded_shape = self.cache["x_padded_shape"]
        _, _, H_pad, W_pad = x_padded_shape
        max_idx = self.cache["max_idx"]
```

Solution: MaxPool2d (backward)

```
dY_flat = dY.reshape(N, C, L)
dX_cols_reshaped = np.zeros(
    (N, C, kH * kW, L), dtype=dY.dtype
)

n_idx = np.arange(N)[:, None, None]
c_idx = np.arange(C)[None, :, None]
l_idx = np.arange(L)[None, None, :]

dX_cols_reshaped[n_idx, c_idx, max_idx, l_idx] = dY_flat

dX_cols = dX_cols_reshaped.reshape(N, C * kH * kW, L)
dX_pad = col2im_batch(
    dX_cols, C, H_pad, W_pad, kH, kW, stride=s
)

if p > 0:
    dX = dX_pad[:, :, p:-p, p:-p]
else:
    dX = dX_pad

return dX
```

Conv2d: naive implementation (per sample)

Goal of Conv2d:

- Apply a bank of filters (kernels) to the input
- Each filter slides over (H, W) and produces one output channel
- Operate on minibatches: $x \in \mathbb{R}^{N \times C_{\text{in}} \times H \times W}$

Naive implementation (what we already did):

- Loop over the batch: $n = 0, \ldots, N-1$
- For each $x_n \in \mathbb{R}^{C_{\text{in}} \times H \times W}$:
  - Apply `im2col` (single-image version)
  - Multiply by the reshaped weights
  - Reshape back to $(C_{\text{out}}, H_{\text{out}}, W_{\text{out}})$

Note: in these slides we implement Conv2d without bias for simplicity, focusing on the batched `im2col` trick.

Problem: the outer loop over N is still in Python, which becomes slow for large batches.

Conv2d as matrix multiplication (single image)

For a **single** input image $x \in \mathbb{R}^{C_{\text{in}} \times H \times W}$:

1. Apply `im2col`:

$$X_{\text{col}} \in \mathbb{R}^{(C_{\text{in}} k_H k_W) \times L}, \quad L = H_{\text{out}} W_{\text{out}}.$$

2. Reshape the weights $W \in \mathbb{R}^{C_{\text{out}} \times C_{\text{in}} \times k_H \times k_W}$ to

$$W_{\text{row}} \in \mathbb{R}^{C_{\text{out}} \times (C_{\text{in}} k_H k_W)}.$$

3. Compute

$$Y_{\text{col}} = W_{\text{row}} X_{\text{col}} \in \mathbb{R}^{C_{\text{out}} \times L}.$$

4. Reshape Y_{col} to $Y \in \mathbb{R}^{C_{\text{out}} \times H_{\text{out}} \times W_{\text{out}}}$.

This is already efficient for a *single* image, but we still repeat it N times in a Python loop.

Batched `im2col` for Conv2d

Idea: apply the same `im2col` trick to the **whole batch**.

Input:

$$x \in \mathbb{R}^{N \times C_{\text{in}} \times H \times W}$$

Batched `im2col`:

$$x \longrightarrow X_{\text{col}} \in \mathbb{R}^{N \times (C_{\text{in}} k_H k_W) \times L}, \quad L = H_{\text{out}} W_{\text{out}}.$$

Interpretation:

- For each sample n, the slice $X_{\text{col}}[n]$ is the usual single-image `im2col`.
- Instead of looping over n, we compute all of them in one vectorized operation.

Once we have X_{col}, we can reorganize it to perform a single large matrix multiplication with the weights.

Conv2d batched: forward (concept)

Shapes:

$$X_{\text{col}} \in \mathbb{R}^{N \times K \times L}, \quad K = C_{\text{in}} k_H k_W, \quad L = H_{\text{out}} W_{\text{out}}.$$

$$W_{\text{row}} \in \mathbb{R}^{C_{\text{out}} \times K}.$$

Strategy:

1. Reshape and transpose X_{col} to

$$X_{\text{mat}} \in \mathbb{R}^{K \times (NL)}$$

   by stacking all patches from all samples.

2. Compute

$$Y_{\text{mat}} = W_{\text{row}} X_{\text{mat}} \in \mathbb{R}^{C_{\text{out}} \times (NL)}.$$

3. Reshape Y_{mat} back to

$$Y \in \mathbb{R}^{N \times C_{\text{out}} \times H_{\text{out}} \times W_{\text{out}}}.$$

No explicit loop over N: the batch dimension is folded into the big matrix.

Conv2d batched: backward (high-level idea)

We need gradients w.r.t.:

$$W \quad \text{and} \quad X.$$

Using the matrix view:

$$Y_{\text{mat}} = W_{\text{row}} X_{\text{mat}}$$

with

$$W_{\text{row}} \in \mathbb{R}^{C_{\text{out}} \times K}, \quad X_{\text{mat}} \in \mathbb{R}^{K \times (NL)}.$$

Gradients:

- Given $\dfrac{\partial \mathcal{L}}{\partial Y_{\text{mat}}}$, the gradient w.r.t. weights:

$$\frac{\partial \mathcal{L}}{\partial W_{\text{row}}} = \frac{\partial \mathcal{L}}{\partial Y_{\text{mat}}} X_{\text{mat}}^{\top}.$$

- The gradient w.r.t. input columns:

$$\frac{\partial \mathcal{L}}{\partial X_{\text{mat}}} = W_{\text{row}}^{\top} \frac{\partial \mathcal{L}}{\partial Y_{\text{mat}}}.$$

Then we reshape $\frac{\partial \mathcal{L}}{\partial X_{\text{mat}}}$ back to (N, K, L) and apply batched `col2im` to obtain $\frac{\partial \mathcal{L}}{\partial X}$.

Exercise: Conv2d batched (overview)

Goal:

- Replace the naive Conv2d implementation (looping over the batch dimension) with a **batched** version.
- Keep the public API unchanged:

$$x \in \mathbb{R}^{N \times C_{\text{in}} \times H \times W} \;\Rightarrow\; y \in \mathbb{R}^{N \times C_{\text{out}} \times H_{\text{out}} \times W_{\text{out}}}.$$

- Use the same idea as for MaxPool2d: apply `im2col` to the whole batch at once.

Key steps:

1. Implement or reuse a batched `im2col` / `col2im`.
2. Rewrite `Conv2d.forward` without loops over N.
3. Rewrite `Conv2d.backward` using the matrix view of the convolution.

Exercise: Conv2d batched – forward

Task: remove the Python loop over the batch in `forward` and use a batched `im2col` instead.

Starting point (simplified):

```
class Conv2d:
    def __init__(self, in_channels, out_channels,
                 kernel_size, stride=1, padding=0):
        # same as before...
        self.in_channels = in_channels
        self.out_channels = out_channels
        self.kernel_size = (kernel_size, kernel_size) \
            if isinstance(kernel_size, int) else kernel_size
        self.stride = stride
        self.padding = padding
        kH, kW = self.kernel_size
        self.weight = np.random.randn(
            out_channels, in_channels, kH, kW
        ) * 0.01
        self.grad_weight = np.zeros_like(self.weight)
```

Exercise: Conv2d batched – forward

```
def forward(self, x):
    # x: (N, C_in, H, W)
    N, C_in, H, W = x.shape
    kH, kW = self.kernel_size
    p = self.padding
    s = self.stride

    if p > 0:
        x = np.pad(
            x,
            ((0, 0), (0, 0), (p, p), (p, p)),
            mode="constant",
        )

    _, _, H_pad, W_pad = x.shape
    H_out = (H_pad - kH) // s + 1
    W_out = (W_pad - kW) // s + 1
    L = H_out * W_out

    # TODO:
    # 1) apply batched im2col: X_col shape (N, K, L)
    # 2) reshape weights to W_col shape (C_out, K)
    # 3) build X_mat shape (K, N*L) and compute Y_mat
    # 4) reshape back to (N, C_out, H_out, W_out)

    # remember to cache what you need for backward:
    # self.X_col, self.x_shape, self.H_pad, self.W_pad, ...
```

Exercise: Conv2d batched – backward (math)

Matrix view of the batched convolution:

$$Y_{\text{mat}} = W_{\text{row}} X_{\text{mat}},$$

where

$$W_{\text{row}} \in \mathbb{R}^{C_{\text{out}} \times K}, \quad X_{\text{mat}} \in \mathbb{R}^{K \times (NL)},$$

and

$$K = C_{\text{in}} k_H k_W, \quad L = H_{\text{out}} W_{\text{out}}.$$

Gradients:

$$\frac{\partial \mathcal{L}}{\partial W_{\text{row}}} = \frac{\partial \mathcal{L}}{\partial Y_{\text{mat}}} X_{\text{mat}}^\top,$$
$$\frac{\partial \mathcal{L}}{\partial X_{\text{mat}}} = W_{\text{row}}^\top \frac{\partial \mathcal{L}}{\partial Y_{\text{mat}}}.$$

Then:

- reshape $\frac{\partial \mathcal{L}}{\partial X_{\text{mat}}}$ back to (N, K, L),
- apply batched `col2im` to obtain $\frac{\partial \mathcal{L}}{\partial X}$ with shape (N, C_{in}, H, W).

Exercise: Conv2d batched – backward (code sketch)

```python
def backward(self, dY):
    # dY: (N, C_out, H_out, W_out)
    N, C_out, H_out, W_out = dY.shape
    L = H_out * W_out
    kH, kW = self.kernel_size
    C_in = self.in_channels
    s = self.stride
    p = self.padding

    # X_col: (N, K, L) saved in forward
    X_col = self.X_col
    K = C_in * kH * kW

    # TODO:
    # 1) build dY_mat shape (C_out, N*L)
    # 2) build X_mat shape (K, N*L)
    # 3) compute dW_row and reshape to self.grad_weight
    # 4) compute dX_mat, reshape back to (N, K, L)
    # 5) apply batched col2im to get dX_pad
    # 6) crop padding (if p > 0) to return dX
```

Exercise: Conv2d batched – hints on shapes

Useful shapes:

- X_{col}: (N, K, L) with

$$K = C_{\text{in}} k_H k_W, \quad L = H_{\text{out}} W_{\text{out}}.$$

- W_{row}: (C_{out}, K).
- X_{mat}: (K, NL), built from X_{col}.
- Y_{mat}: (C_{out}, NL).
- dY_{mat}: same shape as Y_{mat}.
- dX_{mat}: $(K, NL) \Rightarrow$ reshaped back to (N, K, L).

Suggestion: implement and debug the forward pass first, then add the backward pass and compare your gradients against the naive implementation on small random tensors.

Solution: Conv2d batched (init)

```python
class Conv2d:
    def __init__(self, in_channels, out_channels,
                 kernel_size, stride=1, padding=0):
        if isinstance(kernel_size, int):
            kernel_size = (kernel_size, kernel_size)
        self.in_channels = in_channels
        self.out_channels = out_channels
        self.kernel_size = kernel_size
        self.stride = stride
        self.padding = padding

        kH, kW = self.kernel_size
        self.weight = np.random.randn(
            out_channels, in_channels, kH, kW
        ) * 0.01
        self.grad_weight = np.zeros_like(self.weight)
```

Solution: Conv2d batched (forward, part 1)

```python
def forward(self, x):
    # x: (N, C_in, H, W)
    N, C_in, H, W = x.shape
    kH, kW = self.kernel_size
    p = self.padding
    s = self.stride

    self.x_shape = x.shape

    if p > 0:
        x_padded = np.pad(
            x,
            ((0, 0), (0, 0), (p, p), (p, p)),
            mode="constant",
        )
    else:
        x_padded = x

    _, _, H_pad, W_pad = x_padded.shape
    H_out = (H_pad - kH) // s + 1
    W_out = (W_pad - kW) // s + 1
```

Solution: Conv2d batched (forward, part 2)

```
self.H_pad = H_pad
self.W_pad = W_pad
self.H_out = H_out
self.W_out = W_out

W_col = self.weight.reshape(self.out_channels, -1)
self.W_col = W_col

X_col = im2col_batch(x_padded, kH, kW, stride=s)
self.X_col = X_col

N2, K, L = X_col.shape
assert N2 == N
assert K == C_in * kH * kW
assert L == H_out * W_out

X_mat = X_col.transpose(1, 0, 2).reshape(K, N * L)
Y_mat = W_col @ X_mat

out = Y_mat.reshape(
    self.out_channels, N, H_out, W_out
).transpose(1, 0, 2, 3)
return out
```

Solution: Conv2d batched (backward, part 1)

```python
def backward(self, dY):
    # dY: (N, C_out, H_out, W_out)
    N, C_out, H_out, W_out = dY.shape
    kH, kW = self.kernel_size
    C_in = self.in_channels
    s = self.stride
    p = self.padding

    H_pad = self.H_pad
    W_pad = self.W_pad
    W_col = self.W_col
    X_col = self.X_col

    N2, K, L = X_col.shape
    assert N2 == N
    assert K == C_in * kH * kW
    assert L == H_out * W_out
```

Solution: Conv2d batched (backward, part 2)

```python
dY_mat = dY.reshape(N, C_out, L)
dY_mat = dY_mat.transpose(1, 0, 2)
dY_mat = dY_mat.reshape(C_out, N * L)

X_mat = X_col.transpose(1, 0, 2)
X_mat = X_mat.reshape(K, N * L)

dW_col = dY_mat @ X_mat.T
dX_mat = W_col.T @ dY_mat

dX_cols = dX_mat.reshape(K, N, L)
dX_cols = dX_cols.transpose(1, 0, 2)

dX_pad = col2im_batch(
    dX_cols, C_in, H_pad, W_pad, kH, kW, stride=s
)

if p > 0:
    dX = dX_pad[:, :, p:-p, p:-p]
else:
    dX = dX_pad

self.grad_weight = dW_col.reshape(self.weight.shape)
return dX
```

Wrap-up: what we achieved

Conceptual takeaways

- Conv2d and MaxPool2d can be seen as *linear operations* on patches: `im2col` / `col2im` are just a clever reshaping of the data.
- By applying **batched** `im2col`, we turn many small convolutions into a single large matrix multiplication over all samples in the batch.
- MaxPool2d backward can be written without loops using stored argmax indices and NumPy advanced indexing.

Engineering takeaways

- The public API of the layers did not change: the training loop stays exactly the same.
- Most of the speedup comes from:
  - removing Python loops over N,
  - delegating heavy work to optimized BLAS via `@`.
- Further speed improvements are possible (e.g. faster `im2col`/`col2im`), but we now have a clean and reasonably efficient NumPy-based ConvNet core.

Training Time Comparison (Same Model, Same Data)

Version	Time (hh:mm:ss)	Speed-up vs base
NumPy naive (no batched conv/pool)	00:08:35	$1.0\times$
NumPy batched conv/pool	00:01:33	$\approx 5.5\times$
PyTorch	00:00:10	$\approx 51\times$

- Adding batching and vectorized `Conv2d`/`MaxPool2d` reduces training time from 8m35s to 1m33s ($\approx 5.5\times$ faster).
- A mature framework like PyTorch is still about $9\times$ faster than our batched NumPy version, thanks to highly optimized C/C++ kernels and better use of hardware.
- The *math* is the same, but implementation details and low-level optimizations have a huge impact on performance.

Thanks!