Unit 7 – Applications

Convolutional Neural Networks

- **Reference**: R. Gonzalez, "Deep convolutional neural networks [lecture notes]", *IEEE Signal Processing Magazine*, vol. 35. no.6, Nov. 2018, pp. 79-87.
- Neural networks: They are a subset of the field of artificial intelligence (AI).
  - ✓ Predominant type for multidimensional signal processing: Deep Convolutional neural networks (CNNs)
- The term deep refers generically to networks having from a "few" to several dozen or more convolution layers, and deep learning refers to methodologies for training these systems to automatically learn their functional parameters using data representative of a specific problem/domain of interest.
- CNNs are currently being used in a broad spectrum of application areas, all of which share the common objective of being able to automatically learn features from (typically massive) data bases and to generalize their responses to circumstances not encountered during the learning phase.
- Applications: handwriting recognition, machine-printed character recognition, natural language processing, biometrics (face, retinal identification), visual quality inspection, medical diagnoses, and autonomous vehicle navigation.

CNN Architecture

- CNNs are widely used to process 2-D signals (usually images). CNNs are the approach of choice for addressing complex image recognition tasks.
- CNN architecture: layers of convolution, activation, and pooling. The result is then fed into a fully connected network (FCN). Fig. 1 shows an example for hand-written digit recognition.
  - ✓ FCN purpose: map a set of 2-D features into a class label for each input image.

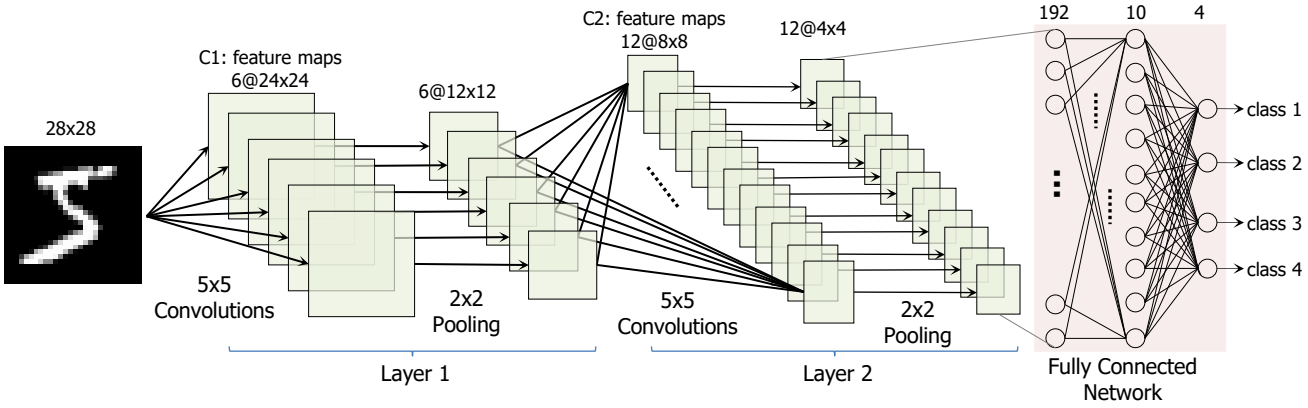

Figure 1. Convolutional Neural Network (CNN) Architecture for hand-written digit recognition. Input: grayscale image. Two convolutional layers. The first layer generates 6 image blocks, while the second layer generates 12 blocks. FCN: 3 layers. Instead of generating a class for each digit, this FCN generates a 4-bit unsigned representation.

Convolutional Layers

- Each convolutional layer is composed of three volumes (each with height, weight, and depth):
  - ✓ Input maps: Input matrices to the layers.
    - First Layer: One matrix (if grayscale or binary image), or 3 matrices (if using RGB color space).
    - Other Layers: The Input maps are the outputs of the previous layers.
  - ✓ Feature maps: A feature map results from adding up all convolutions between each input map and an associated kernel, followed by two pixel-to-pixel operations: adding a bias (this is optional) and applying an activation function. Each feature map has an associated set of kernels (as many kernels as input maps) and a scalar bias. From Fig.1, we have:
    - First Layer (grayscale image): One input map and 6 feature maps. Each feature map has an associated kernel (as there is only one input map) and scalar bias. A feature map is the result of one 2-D convolution, plus adding the bias to each pixel and then applying an activation function to each pixel.
    - Second Layer: 6 input maps and 12 feature maps. Each feature map has an associated set of 6 kernels; there is a total of 6x12 kernels. A feature map is the result of applying 6 2-D convolutions, adding them up, followed by adding a bias to each pixel and applying an activation function to each pixel.
  - ✓ Pooled feature maps (or pooled maps, output maps): Also called sub-sampling or down-sampling. It reduces the size of the feature maps. Not always used in some cases. Types: max, average, sum.

- The size of the 2D-arrays generally varies from volume to volume. However, all maps within a volume are of the same size.

- Fig. 2 depicts how the feature maps and output (pooled) maps are computed for Convolutional Layer 1 (C1) and Convolutional Layer 2 (C2). Each feature map has an associated kernel (or set of kernels), and a bias. The activation function is the same for all feature maps in a Convolutional Layer.

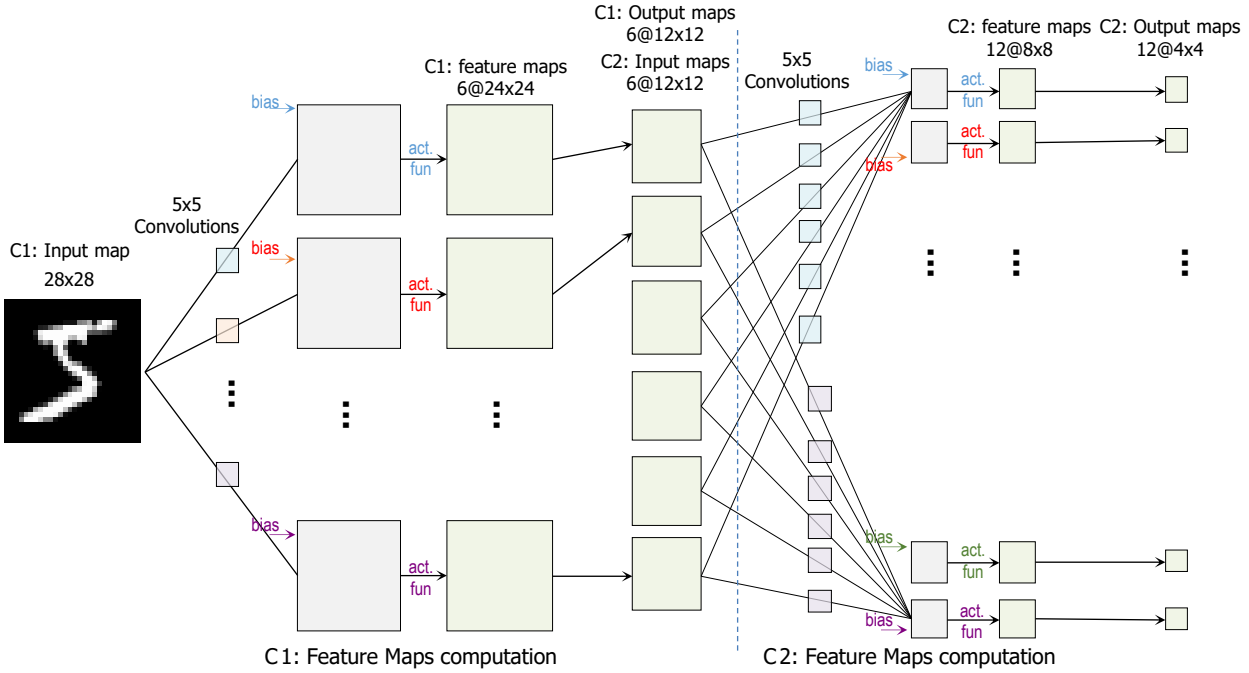

Figure 2. Computation of Feature Maps for C1 and C2 (see CNN in Fig. 1). C1: a feature map is the result of one 2-D convolution, plus the scalar bias and the activation function. C2: a feature map is the result of a sum of six 2-D convolutions, plus the scalar bias and the activation function.

CONVOLUTION

- The primary purpose of convolution is to extract features from the input image (e.g.: edges, orientations), hence the term feature maps for the outputs of a convolution layer.
- 2-D filters (kernels) have the ability to detect/emphasize specific features in images. Example: edges, orientations.
- This is a computation-intensive and popular application. The input image (I) of size SX×SY (SX columns, SY rows) is convolved with a kernel (K) of size KX×KY to generate and output image (O) of size (SX+KX-1)×(SY+KY-1).

$$O(m,n) = I(m,n) * K(m,n) \sum_{j=0}^{SX-1} \left(\sum_{i=0}^{SY-1} I(i,j) \times K(m-i, n-j) \right)$$

- m = 0, …, SY+KY-1, n = 0, …, SX+KX-1.
  - ✓ For I, the bounds are: i = 0, …, SY-1, j = 0, …, SX-1.
  - ✓ For K, the bounds are: i = 0, …, KY-1, j = 0, …, KX-1. During computation, there will be indices of K(m-i,n-j) that fall outside the boundaries of K, meaning that the product term is ignored (i.e., each O(m,n) requires at most KX×KY computations). In addition, this means that when computing O(m,n) at the borders, we can graphically visualize the computation where I(i,j) as zero-padded outside the boundaries of I.

- Fig. 3 illustrates these concepts. The convolution operation is like a sliding window: for every $O(m,n)$, the flipped kernel overlaps with I, where we multiply-and-add only the overlapping elements. In some cases, there are elements of the flipped kernel that do not overlap with the elements of I. Here, these operations do not occur (this is like I was zero-padded).
  - ✓ Fig. 3(a): Two kernels will be used to illustrate the operation. Note how the kernel is flipped. The kernel is usually square. We use one kernel with odd sizes and the other with even sizes.
  - ✓ Fig. 3(b): Kernel with odd sizes. The location of an output value O(m,n) overlaps with the center of the kernel, indicated as a red square.
  - ✓ Fig. 3(c): Kernel with even sizes. The location of an output value O(m,n) overlaps with an element of the kernel, indicated as a red square located in the position $(\lfloor KX/2 \rfloor + 1, \lfloor KY/2 \rfloor + 1)$ of the kernel. Kernel indices: (1,1) to (KX, KY).

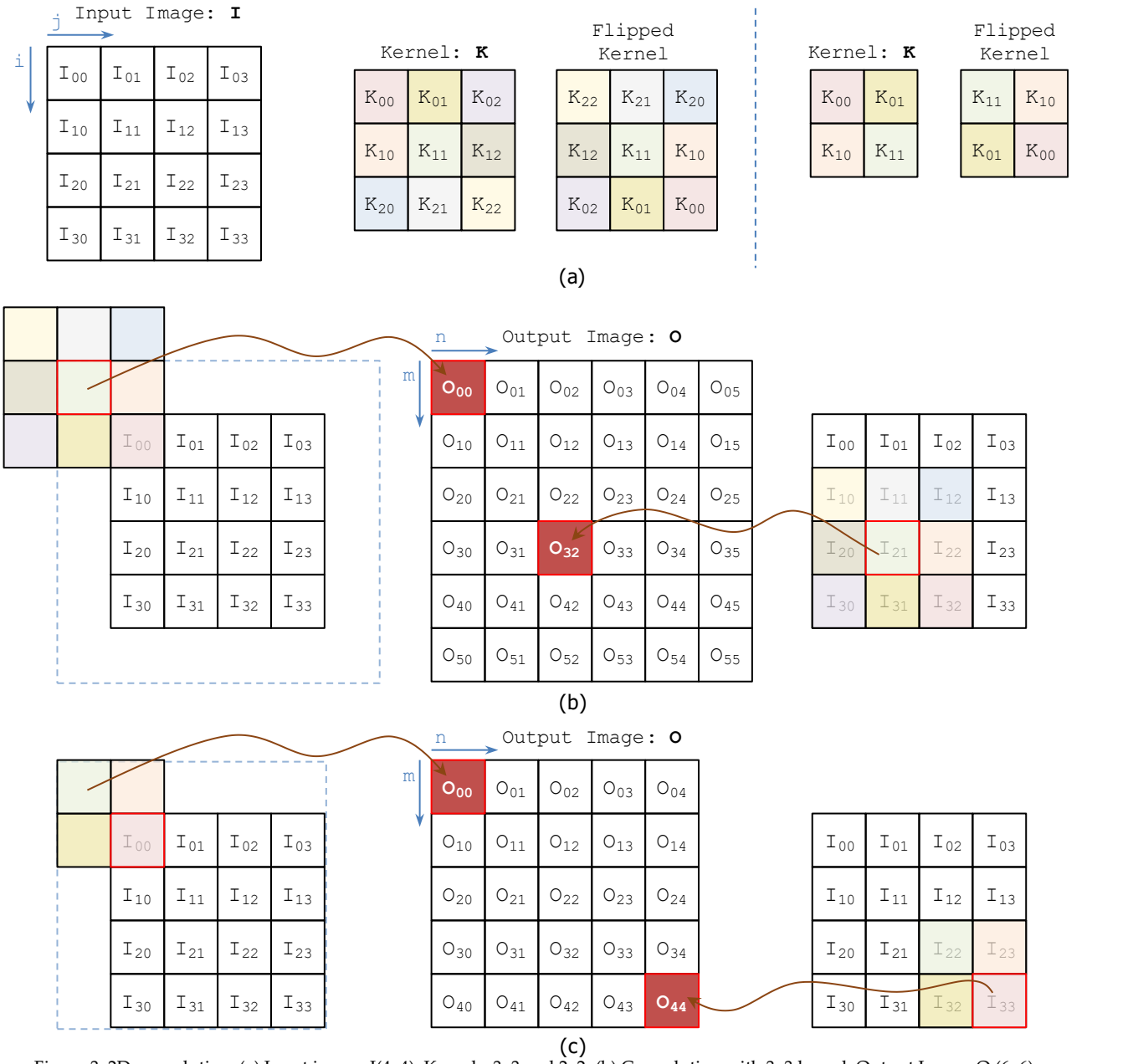

Figure 3. 2D convolution. (a) Input image: I(4x4). Kernels: 3x3 and 2x2. (b) Convolution with 3x3 kernel. Output Image: O (6x6). (c) Convolution with 2x2 kernel. Output image: O (5x5).

- Output matrix size. Fig. 4 depicts the three approaches for SX=SY=4, KX=KY=3.
  - ✓ Full convolution: It returns a matrix of size (SX+KX-1)×(SY+KY-1).
  - ✓ Central part of the convolution (SX×SY): The output matrix size is that of the input I. If KX or KY is even, then the 'center' leaves one more at the beginning than the end. You can use the following formula to get the central part of the convolution out of the full convolution of size (SX+KX-1)×(SY+KY-1). The indices consider the full convolution output to be from (1,1) to (SX+KX-1,SY+KY-1):

$$\left\lfloor \frac{KX}{2} \right\rfloor + 1 : SX + \left\lfloor \frac{KX}{2} \right\rfloor, \left\lfloor \frac{KY}{2} \right\rfloor + 1 : SX + \left\lfloor \frac{KY}{2} \right\rfloor$$

  - ✓ Narrow convolution (SX-KX+1)×(SY-KY+1): Convolution is computed only in those positions where all the kernel values have a matching input component. This is not the case at the boundaries of the input. This is the preferred approach in Convolutional Layers.

- Note that each convolution element can be thought of as the result of a dot product. Also, sometimes, a scalar value (bias b) is added to each pixel resulting from convolution.
  - ✓ Since each output value computation is essentially similar to how the neuron output value is generated (see FCN section below), we can apply similar techniques (e.g.: back-propagation) to train the convolutional layer parameters.

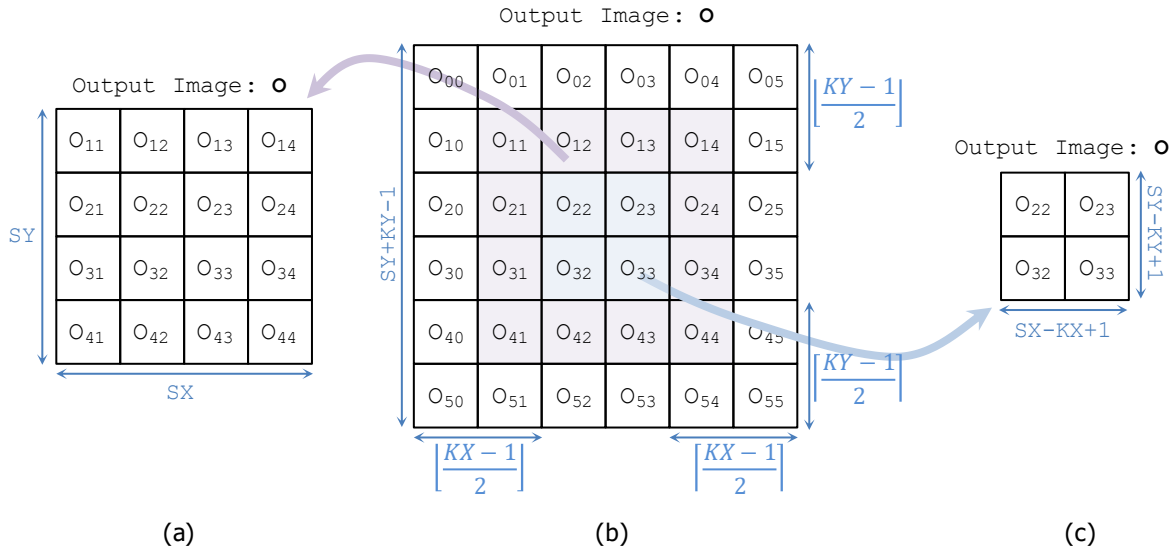

Figure 4. Different approaches for selecting output matrix size. I (4x4), K(3x3). (a) Central part of the convolution: O(4x4). This is common in image filtering applications. (b) Full convolution: O(6x6). This is what is computed by the convolution formula. (c) Narrow convolution: O(2x2). This is preferred in convolutional neural network applications.

- **Note**: The Convolutional Layers in CNNs do not flip the kernel prior to performing the sliding window multiply-and-add computation. This operation is actually called 'cross-correlation'. However, since the computations are essentially identical we still use the 'convolution' label.

ACTIVATION FUNCTION (introducing non-linearity)

- We apply an activation function to each pixel. This is a pixel-to-pixel operation. Common activation functions include the following non-linear operations:
  - ✓ Rectified Linear Unit (ReLU): $\sigma(z_j^l) = \max(0, z_j^l)$
  - ✓ Hyperbolic Tangent: $\sigma(z_j^l) = \tanh(z_j^l)$
  - ✓ Sigmoid function: $\sigma(z_j^l) = {}^{1}\!/\!_{\left(1+e^{-z_j^l}\right)}$

POOLING (or Sub-sampling)

- This operation reduces the dimensionality of each feature maps but retains the most important information. Spatial pooling can be of different types: max, average, sum, etc.
- For Max Pooling we define a spatial neighborhood (e.g., 2x2 window) and take the largest element within that window. Instead of taking the largest element, we could also take the average (Average Pooling) or sum of all elements in that window (Sum Pooling). In practice, Max Pooling has been shown to work better.
- Note that the pooling operation is applied separately to each feature map.
- A consequence of pooling is significant data reduction, which helps speed up processing.
- Fig. 4 shows an example of Max Pooling operation by using a 2x2 window that slides by 2 cells.

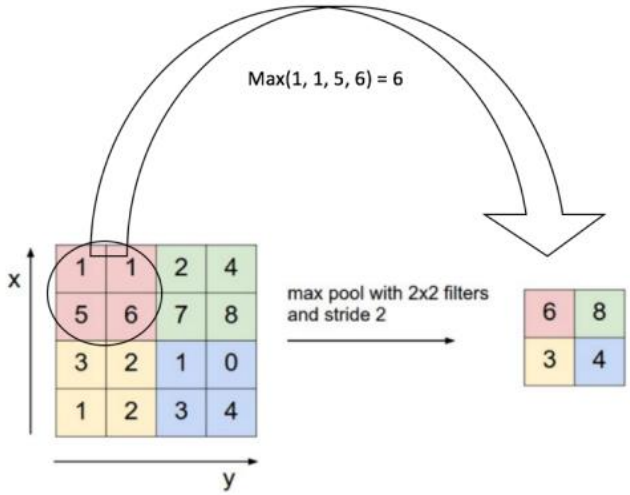

Figure 5. Max Pooling operation. Window size: 2x2. Stride: 2

FULLY CONNECTED NETWORK (FCN)

- The purpose of an FCN is to map a set of features into a class label for each input sample (e.g. image).
- A 3-layer neural network (also called a Fully Connected Layer) is depicted in Fig. 6(a). Fig. 6(b) depicts the inputs and output of the first neuron (index '1') in layer 3.
  - ✓ Input layer ($l = 1$): It represents the input values to the network. There is no computation in Layer 1.
- Fig. 6(c) depicts an artificial neuron model. The neuron output (action potential a_j^l) results from applying an activation function to the membrane potential (z_j^l). The index corresponds to neuron j in layer l.
- The membrane potential z_j^l is a dot product between the inputs and the associated weights, to which a bias is then added.

$$z_j^l = \sum_k w_{jk}^l a_k^{l-1} + b_j^l, l > 1$$

- The action potential intensity of a neuron is denoted by a_j^l, and it is modeled as a scalar function (activation function) of z_j^l:

$$a_j^l = \sigma(z_j^l) = \sigma\left(\sum_k w_{jk}^l a_k^{l-1} + b_j^l\right), l > 1$$

  - Common activation functions include the following non-linear operations:
    - ✓ Rectified Linear Unit (ReLU): $\sigma(z_j^l) = \max(0, z_j^l)$
    - ✓ Hyperbolic Tangent: $\sigma(z_j^l) = \tanh(z_j^l)$
    - ✓ Sigmoid function: $\sigma(z_j^l) = 1 / \left(1 + e^{-z_j^l}\right)$

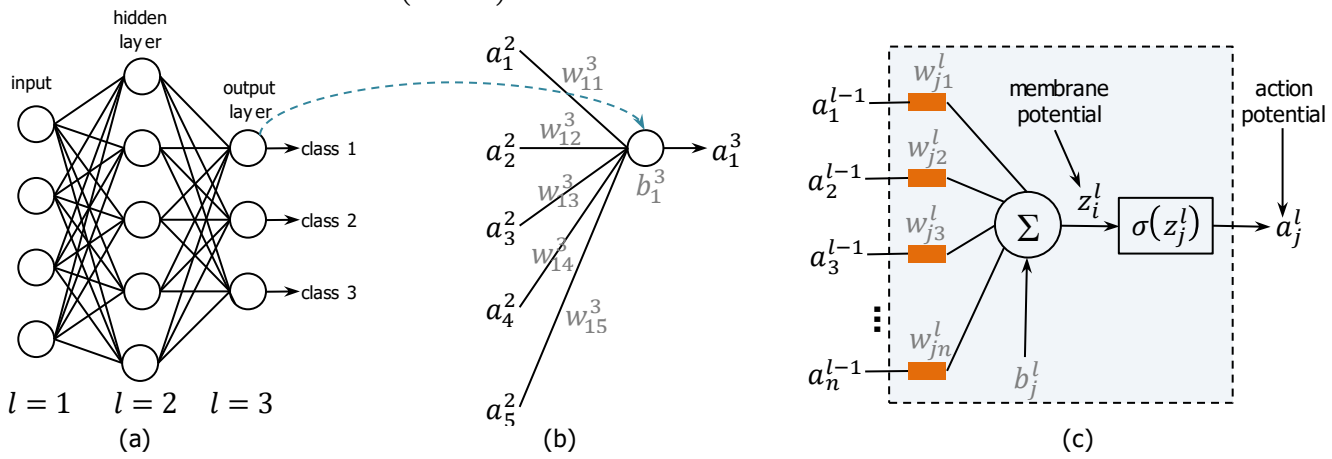

Figure 6. (a) 3-layer neural network. (b) First neuron (index '1') in layer l=3. (c) Artificial neuron model. The membrane potential is a sum of products (input activations by weights) to which a bias term is added. The index corresponds to neuron j in layer l. The input activations come from a previous layer (l-1).

- The output of a layer l can be described using a vectorized notation:

$$a^l = \sigma(z^l), \qquad z^l = w^l a^{l-1} + b^l, l > 1$$

Where:
w^l: weight matrix (NO rows by NI columns) of the layer l,
a^{l-1}: action potential vector (NI rows) of the previous layer l-1.
b^l: bias vector (NO rows) of the layer l.
z_l: membrane potential vector (NO rows) of the layer l.
a^l: action potential vector (NO rows) of the layer l.

- Fig. 7 depicts the matrix operation for z^l. NO: # of neurons in layer l. NI: # of inputs for each neuron in layer l.

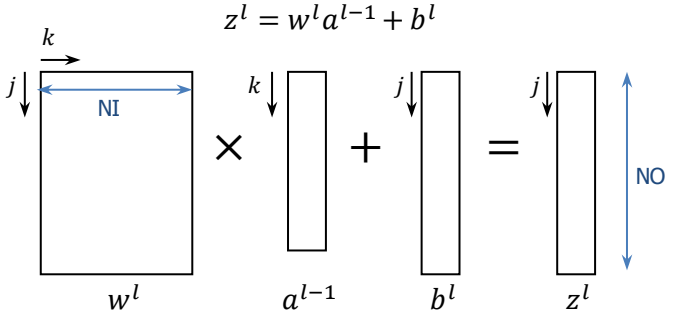

Figure 7. Matrix operation for the computation of all membrane potentials in layer l.

C++/TBB CODE

FCN: COMPARISON OF DIFFERENT APPROACHES

- FCN (4 layers, LT=4, L=3): First layer is just inputs.
  - ✓ Layer l=0 (input): 28x28 = 784 outputs
  - ✓ Layer l=1: 784 inputs, 120 outputs (ReLU)
  - ✓ Layer l=2: 120 inputs, 84 outputs (ReLU)
  - ✓ Layer l=3: 84 inputs, 10 outputs. Classified output: the one with the maximum value.
- Testing Dataset: 10,000 samples of 784 (28x28) elements each.
- Trained model available in Python, called 'pytst'. Accuracy: 95.01%

TABLE I. COMPARISON OF VARIOUS PARALLEL APPROACHES FOR FCN IMPLEMENTATION

Code Style	Comments	Execution Time 'pytst'
`seq\nnetwork.cpp`	Straightforward sequential implementation	1.67s
`tbb\nnetwork.cpp`	At each layer, all action potentials are computed in parallel via *parallel_for*. Each computation: dot product, plus bias, and ReLU. parallel_reduce: tried for each dot product, but it increased computation time by ~7X.	1.24s
`tbb\nnetwork_a.cpp`	We generated several FCNs (copying the object PN times) so that they can compute data in parallel. PN: # of samples computed concurrently. This includes the *parallel_for* approach specified in nnetwork.cpp.	PN=2: 1.67s PN=4: 1.67s PN=8: 1.65s
`tbb\nnetwork_b.cpp`	Similar to nnetwork_b.cpp, but code is optimized: the pointers to weights and biases is reused. Faster, though PN does not really affect the speed (PN=100 resulted in ~ time).	PN=2: 0.95s PN=4: 0.95s PN=8: 0.93s

CNN: COMPARISON OF DIFFERENT APPROACHES

- CNN:

Conv Network (2 layers, LC = 2)	FCN (4 layers, LT=4, L=3). First layer is just inputs.
✓ Layer 1: 1 input channel, 6 output channels, 6x1 3x3 kernels + 6x1 bias, 2x2 max pool. ▫ Input Channel: size of inputs: 28x28=784 pixels. ✓ Layer 2: 6 input channels, 16 output channels, 16x6 3x3 kernels + 16x1 bias, 2x2 max pool.	✓ Layer l=0 (input): 16x5x5 = 400 outputs ✓ Layer l=1: 400 inputs, 120 outputs (ReLU) ✓ Layer l=2: 120 inputs, 84 outputs (ReLU) ✓ Layer l=3: 84 inputs, 10 outputs. Classified output: the one with the maximum value.

- Testing Dataset: 10,000 samples of 784 (28x28) elements each.
- Trained model available in Python, called 'pycnn'. Accuracy: 96.97%

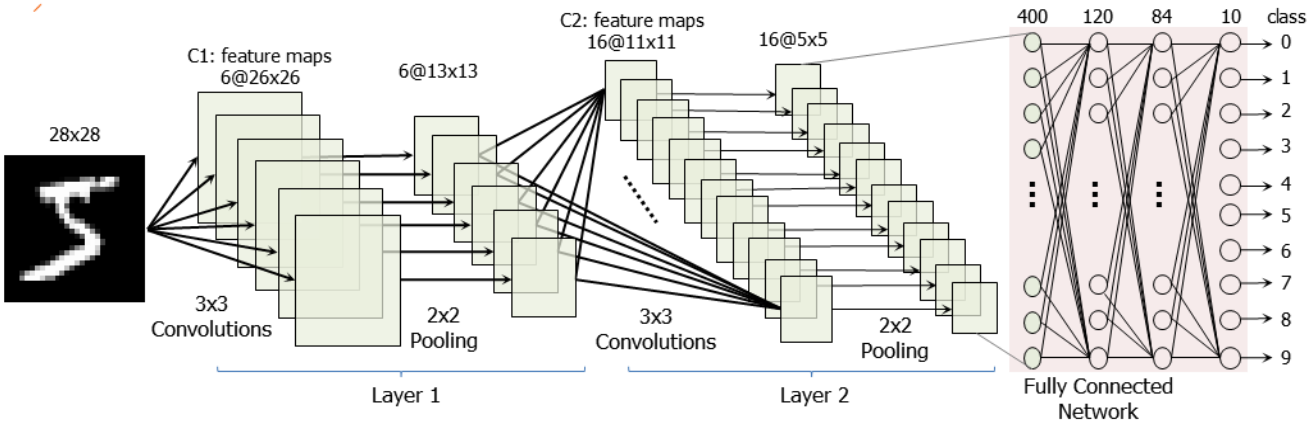

Figure 8. CNN Architecture associated with the results of Table II. The FCN Architecture can be associated with the results of Table I (if the first layer has 784 inputs instead of 400)

TABLE II. COMPARISON OF VARIOUS PARALLEL APPROACHES FOR CNN IMPLEMENTATION

Code Style	Comments	Execution Time 'pycnn'
`seq\mycnn.cpp`	Straightforward sequential implementation (both convolutional network and FCN)	5.88 s
`tbb\mycnn.cpp`	▪ Convolutional Network: At each layer, all output features are computed in parallel via *parallel_for*. Each computation: sum of convolutions, plus bias, plus ReLU, plus pooling. ✓ parallel_reduce: tried for the sum of convolutions (Layer 2 only, Layer 1 not needed as there is only one convolution per output feature). Computation time barely affected. ▪ FCN: like \tbb\nnetwork.cpp	3.44 s *parallel_reduce:* 3.40 s

`tbb\my_cnn_p.cpp`	▪ Convolutional Network and FCN: Like tbb/mycnn.cpp. ▪ parallel_pipeline: Computations divided into all <u>serial</u> stages: ✓ Approach 'a': 3 stages. (PIP_APP = 0) ▫ Stage 1: Issue pointers to image locations in a linear array (i.e., issue linear images) ▫ Stage 2: Convert linear image into a 2D array, then apply Conv Network. ▫ Stage 3: Convert pooled feature maps (2D arrays) into a single linear array, then apply FCN. ✓ Approach 'b': 3 stages. (PIP_APP = 1) ▫ Stage 1: Issue images as pointers to 2D arrays (converts linear array into 2D array) ▫ Stage 2: Apply Conv Network. ▫ Stage 3: Convert pooled feature maps (2D arrays) into a single linear array, then apply FCN. ✓ Approach 'c': 4 stages. (PIP_APP = 2) ▫ Stage 1: Issue pointers to image locations in a linear array (i.e., issue linear images) ▫ Stage 2: Convert linear image into a 2D array, then apply Conv Layer 1. ▫ Stage 3: Apply Conv Layer 2 ▫ Stage 4: Convert pooled feature maps (2D arrays) into a single linear array, then apply FCN.	Approach 'a': 3.37 s Approach 'b': 3.38 s Approach 'c': 3.34 s *parallel_reduce*: Approach 'a': 3.56 s Approach 'b': 3.55 s Approach 'c': 3.55 s
`tbb\mycnn_m.cpp`	▪ Convolutional Network and FCN: Like tbb/mycnn.cpp ▪ We generated several CNNs (copying the object PN times) so that they can compute data in parallel (via parallel_for_. PN: # of samples computed concurrently. (PN > 2 -> no diff)	PN=2: 3.38 s
	▪ Optimized approach: the pointers to weights and biases is reused.	PN=2: 3.22 s

TRAINING

- This is the ability to use sampling training data to learn the operation parameter of each network layer.
- The objective is to determine the weights and biases of the convolutional layers as well as of the neural network layers.
  - ✓ Back-propagation algorithm: Tool to iteratively adjust the network parameters (kernel coefficients, biases, neuron weights, neuron biases).
- MNIST database: 60,000 training samples; 10,000 testing samples. Sample: 28x28 grayscale image

ALGORITHMS FOR MEDIAN FILTER COMPUTATIONS

FORGETFUL_SELECTION

- This algorithm only works for odd-length array; it gets the median out of an array.
  - ✓ Note: the median of an even-length array is the mean of the 2 center elements (once array is sorted). Here, for integer elements, the median may not be integer.
- Input data: $n = k \times k$ pixels (k odd). For k=3,5,7,9,11: k^2 is odd.
- Algorithm: It appeared in [Perrot2014 first] (then in [Salvador2018]).
  - ✓ Take the first r elements of unsorted n-element sequence.
    - $r = \left\lceil \frac{n}{2} \right\rceil + 1$ (Perrot2014), $r = \frac{n+3}{2}$ (Salvador 2018). These formulas match for $k =$ 3,5,7,9,11.
  - ✓ Perform '***partial_forgetful_selection' (pfs)***: Find the extrema (minimum and maximum). Remove these elements from the array.
  - ✓ Insert the remaining $n - r$ elements one by one. For each insertion, perform a '***partial_forgetful_selection***'. Stop when there is only one element left (the median).
  - ✓ Total number of '***partial_forgetful_selections***' (also called steps): $n - r + 1 = n - \left\lceil \frac{n}{2} \right\rceil$ (Perrot2014). One step to process the first r elements, and then $n - r$ steps.

- Note: You cannot get the median by first getting the median of a portion of an n-element vector, then inserting that median into the remaining elements, and then computing the final median out of those elements. The ***forgetful_selection*** algorithm does not do this, it rather performs a ***partial_forgetful_selection*** on the first r elements (leaving $r - 2$ elements), and then inserts the remaining elements (one by one) performing a ***partial_ forgetful_selection*** every time.

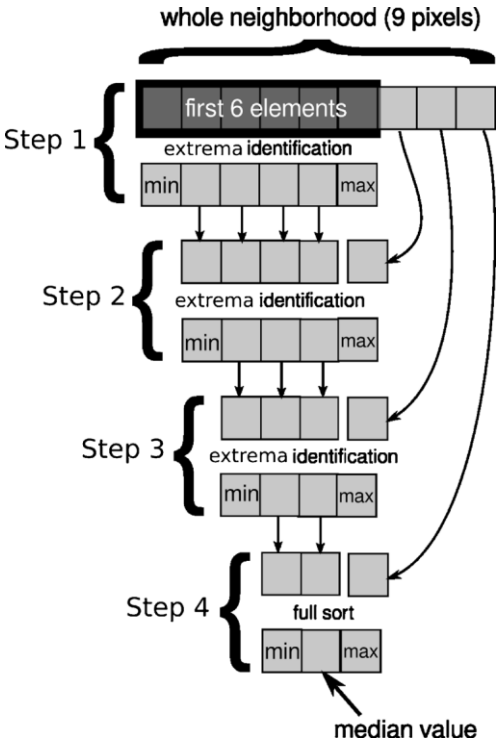

Figure 1. ***forgetful_selection*** for $n = 9$.

PARALLEL COMPUTATION OF 4 PIXELS

- Simple approach: feed 4 $k \times k$ matrices (adjacent, as per a median filter operation), and compute the median of each of them in parallel. Since the matrices are adjacent, there are redundant computations that can be exploited (that are not used here).
- [Salvador2018 approach]: feed one $(k + 1) \times (k + 1)$ matrix. This is like feeding 4 $k \times k$ matrices (adjacent). We get the median out of each of the $k \times k$ matrices while overlapping some computations (redundancies exploited). k odd, $k = 5,7,9,11$

[SALVADOR2018] APPROACH

- Here, we feed a matrix A of size $(k + 1) \times (k + 1)$, and generate 4 output pixels. $i = 1,2,3,4$ (the median out of each $k \times k$ matrix)
- Window of interest for each output pixel: This is its corresponding $k \times k$ matrix.
  - ✓ Size of window of interest for each output pixel: $s = k \times k$.
  - ✓ Indices (starting at 1) of windows of interest for each pixel (with respect to matrix A of size $(k + 1) \times (k + 1)$):
    - Pixel 1: $A(1{:}\,k, 1{:}\,k)$
    - Pixel 2: $A(1{:}\,k, 2{:}\,k + 1)$
    - Pixel 3: $A(2{:}\,k + 1, 1{:}\,k)$
    - Pixel 4: $A(2{:}\,k + 1, 2{:}\,k + 1)$

- Common region for all 4 output pixels: $(k - 1) \times (k - 1)$ matrix. This is the center of matrix A of size $(k + 1) \times (k + 1)$.
  - ✓ Size of the common region for all output pixels: $sc = (k - 1) \times (k - 1)$.
  - ✓ Indices of the common region: $A(2{:}\,k, 2{:}\,k)$

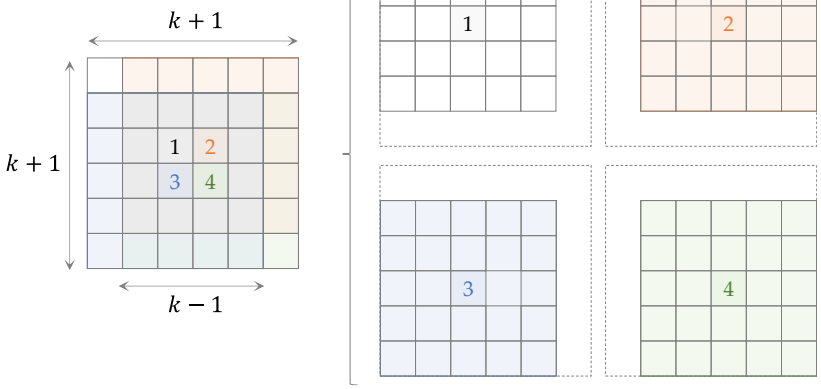

Figure 2. Input matrix A with $(k + 1) \times (k + 1)$ elements ($k = 5$). It also displays the windows of interest for each output pixel ($i = 1,2,3,4$)

- ***Decomposition of the window of interest ($s = k \times k$) for each output pixel***:
  - ✓ Given the $(k+1) \times (k+1)$ input matrix, the window of interest $(k \times k)$ for each output pixel can be decomposed as follows:
    - Common region: $sc = (k-1) \times (k-1)$ pixels
    - Row vector: $k-1$ pixels.
    - Column vector: $k-1$ pixels.
    - Corner element: 1 pixel.
  - ✓ Table I lists the elements to process for each output pixel ($i = 1,2,3,4$):
    - We will first process the common region (sc elements), shared by all the $k \times k$ windows of interest. Then, we process the remaining $s - sc$ elements: row vector, column vector, and corner element.

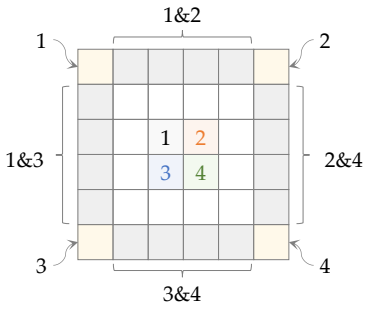

Figure 3. Matrix A of size $(k+1) \times (k+1)$, $k = 5$.. It shows: common region ($(k-1) \times (k-1)$ center matrix), the row vectors (1&2, 3&4), column vectors (1&3, 2&4), and corner elements for the output pixels $i = 1,2,3,4$

TABLE I.DECOMPOSITION OF A WINDOW OF INTEREST FOR EACH OUTPUT PIXEL THIS ALLOWS US TO EXPLOT REDUNDANT COMPUTATIONS

	sc elements	$s - sc$ elements		
	Common region: $(k-1) \times (k-1)$	Row vector: $(k-1)$	Column vector $(k-1)$	Corner element: (1)
Pixel 1	$A(2{:}k, 2{:}k)$	Vector 1&2: $A(1,2{:}k)$	Vector 1&3: $A(2{:}k, 1)$	$A(1,1)$
Pixel 2		Vector 1&2: $A(1,2{:}k)$	Vector 2&4: $A(2{:}k, k+1)$	$A(1, k+1)$
Pixel 3		Vector 1&3: $A(2{:}k, 1)$	Vector 3&4: $A(k+1{:}2{:}k)$	$A(k+1,1)$
Pixel 4		Vector 2&4: $A(2{:}k, k+1)$	Vector 3&4: $A(k+1{:}2{:}k)$	$A(k+1, k+1)$

  - Note that $s = k \times k = (k-1) \times (k-1) + 2 \times (k-1) + 1$, $sc = (k-1) \times (k-1)$, $s - sc = 2 \times (k-1) + 1$
  - We could just perform ***forgetful_selection*** on each of the 4 $k \times k$ matrices (simple parallelization approach).

- **Procedure**: Each $k \times k$ matrix ($i = 1,2,3,4$) has $r = \frac{k^2+3}{2}$. $s = k \times k$, $sc = (k-1) \times (k-1)$, $s - sc = 2 \times (k-1) + 1$
  - ✓ <u>Common region (sc elements)</u>: Perform an incomplete '***forgetful_selection***' on one $k \times k$ matrix (any). Here, we <u>only</u> process the sc common region elements: (Fig. 4 shows an example for $k = 5$).
    - Perform one ***partial_forgetful_selection*** on the first r elements. We will be left with $r - 2$ elements.
    - Insert the remaining $sc - r$ (<u>this is not</u> $s - r$) elements one by one. For each insertion, perform a ***partial_forgetful_selection***.
      - Here, unlike the complete ***forgetful_selection***, after we insert all the $sc - r$ elements, we are left with $r - 2 - (sc - r) = 2r - sc - 2$ elements (we do not get the median). Each time we insert an element, we perform a ***partial_forgetful_selection*** that removes the extrema (two elements). So, there is a net loss of one element per insertion (there are $sc - r$ insertions).
    - So far, we have performed $1 + sc - r$ ***partial_forgetful_selections*** on the sc common elements. The resulting $2r - sc - 2$ elements are valid for all 4 output pixels. This is the redundancy that we exploit, as we only need to do this once.

  - ✓ <u>Remaining elements ($s - sc$ elements per output pixel $i = 1,2,3,4$)</u>: Fig. 5 depicts this procedure for one pixel and $k = 5$.
    - To complete the ***forgetful_selection*** procedure, for each output pixel (with a $k \times k$ region of interest) $i = 1,2,3,4$, do:
      - Use the $2r - sc - 2$ elements (from the previous operation) and insert $s - sc$ elements (one by one) and perform the associated $s - sc$ ***partial_forgetful_selections*** (for a total of $1 + s - r$ ***partial_forgetful_selections***). Result: pixel i (median of the associated $k \times k$ region).
      - Keep in mind: The $s - sc$ elements are different for each output pixel.
    - Since every insertion causes a net loss of 1 element, at the end of the procedure, we are left with $2r - sc - 2 - (s - sc) = 2r - 2 - s$ elements. Since $r = \frac{s+3}{2}$, then $2r - 2 - s = s + 3 - 2 - s = \mathbf{1}$ element (as it should, this is the median).
    - These are 4 independent computations (that could run in parallel).

  - ✓ Table II shows numeric examples for various values of k. Note that the algorithm DOES NOT work for $k = 3$, as first we would need to process $r = 6$ pixels, and then insert $sc - r = -2$ pixels. Thus, for $k = 3$, we just apply 4 normal $k \times k$ ***forgetful_selections***.

TABLE II. NUMBER OF PROCESSED ELEMENTS FOR VARIOUS VALUES OF K.

k	$2r - sc - 2$ elements (valid for all output pixels)	$s - sc$ elements to insert (one by one): different for each pixel			s	r	sc	$sc - r$	$s - sc$
		Row vector: $(k-1)$	Column vector: $(k-1)$	Corner					
3	6	2	2	1	9	6	4	-2	5
5	10	4	4	1	25	14	16	2	9
7	14	6	6	1	49	26	36	10	13
9	18	8	8	1	81	42	64	22	17
11	22	10	10	1	121	62	100	38	21

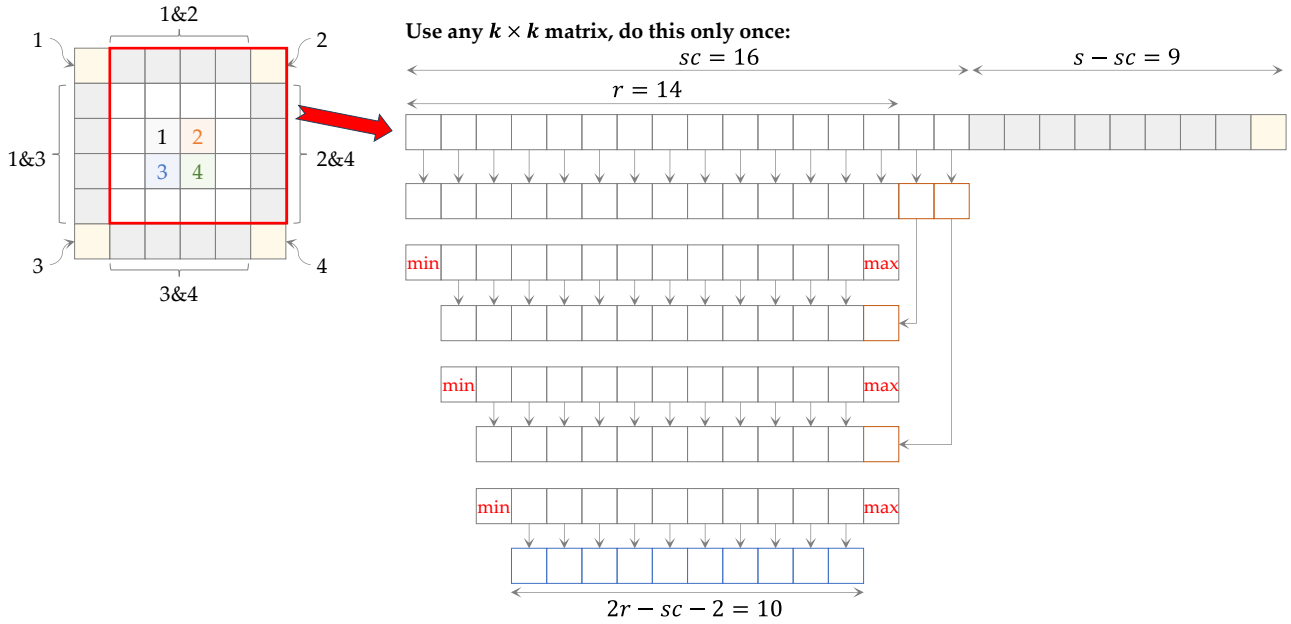

Figure 4. **Part 1** [Salvador2018]: process <u>only</u> the common region (sc elements) with $1 + sc - r$ ***partial_forgetful_selections***. We can use any of the 4 $k \times k$ matrices. The figure uses the $k \times k$ window of interest for $i = 2$.

- We can further save $2 \times (k - 1)$ ***partial_forgetful_selections***.
  - ✓ Pixels 1 and 2 have Vector 1&2 in common. So, we insert this vector ($k - 1$ '*partial forgetful_selections*). The resulting vector is valid for Pixels 1 and 2. Then:
    - Pixel 1: Insert Vector 1&3 and corner element $A(1,1)$, i.e., $k - 1 + 1$ elements (one by one) and perform the corresponding $k - 1 + 1$ ***partial_forgetful_selections***.
    - Pixel 2: Insert Vector 2&4 and corner element $A(1, k + 1)$, i.e., $k - 1 + 1$ elements (one by one) and perform the corresponding $k - 1 + 1$ *'partial forgetful_selections'*.
  - ✓ Pixels 3 and 4 have Vector 3&4 in common. So, we insert this vector ($k - 1$ '*partial forgetful_selections*'). The resulting vector is valid for Pixels 3 and 4. Then:
    - Pixel 3: Insert Vector 1&3 and corner element $A(k + 1,1)$, i.e., $k - 1 + 1$ elements (one by one) and perform the corresponding $k - 1 + 1$ *'partial forgetful_selections'*.
    - Pixel 4: Insert Vector 2&4 and corner element $A(k + 1, k + 1)$, i.e., $k - 1 + 1$ elements (one by one) and perform the corresponding $k - 1 + 1$ *'partial forgetful_selections'*.

- Overall, for $k \neq 3$, this procedure saves (at least in software):
  - ✓ $(1 + sc - r) \times 3$ ***partial_forgetful_selections***. Since we only need to process the common region (sc elements) once.
  - ✓ $2 \times (k - 1)$ ***partial_forgetful_selections***: As explained above, besides the common region, pixels have a row or column vector in common.

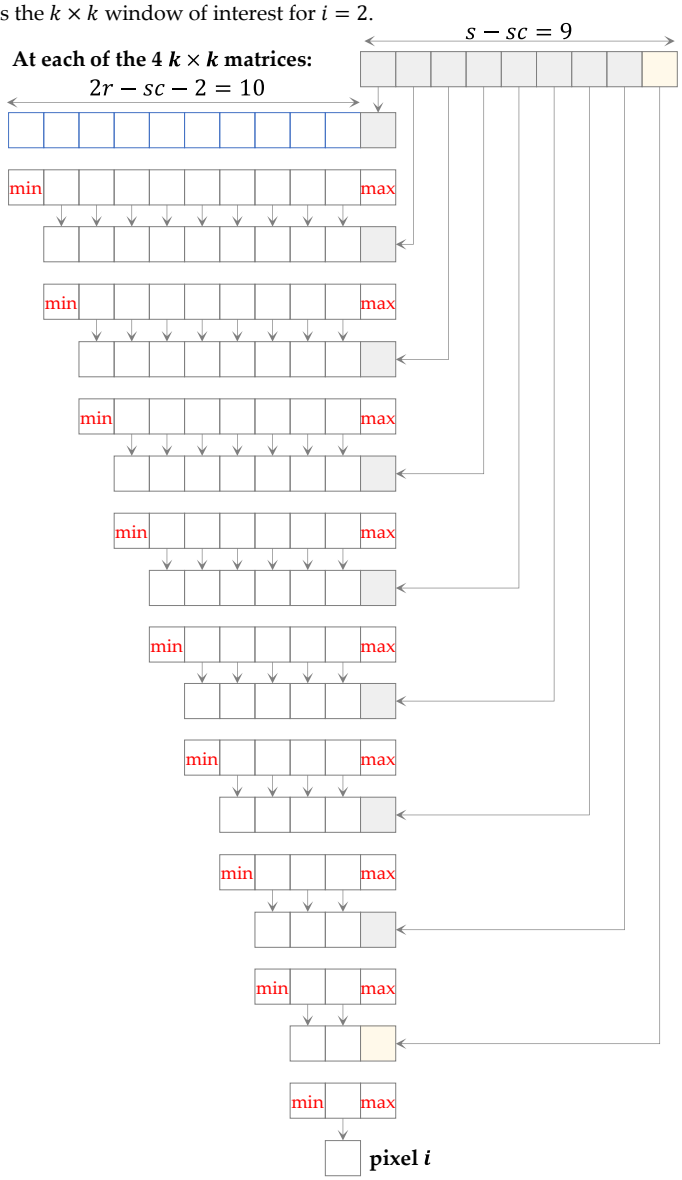

Figure 5. **Part 2** [Salvador2018]: Insert the remaining $s - sc$ pixels (perform $s - sc$ ***partial_forgetful_selections***) and get output pixel i (the median). Repeat this procedure for each one of the output pixels (with associated $k \times k$ matrix). The figure depicts an example for $k = 5$.

ADAPTIVE BEAMFORMING

- Widely used in radar, sonar, speech acquisition, and mobile/wireless communication.
- Adaptive Beamformers are the main component in a switched-beam smart antenna.
  - ✓ Switched-beam smart antenna: system that can select from one of many predefined beam patterns in order to emphasize the level of signals of interest while minimizing undesired signals.

BEAMFORMING STRUCTURE

- This is depicted in Fig. C.1. An array of M sensors generates a snapshot (collection of M complex samples) $\boldsymbol{y}(n)$ at time n.
  - ✓ Output signal $z(n)$: product of the snapshot $\boldsymbol{y}(n)$ (collection of signals from M antenna elements) by a complex weighting vector. Goal: to suppress undesired signals and to emphasize signals from desired directions of arrivals. The weights are adaptively adjusted to suppress time-varying undesired signals with unknown direction of arrival. The maximum gain should be achieved in the direction of the desired signal.

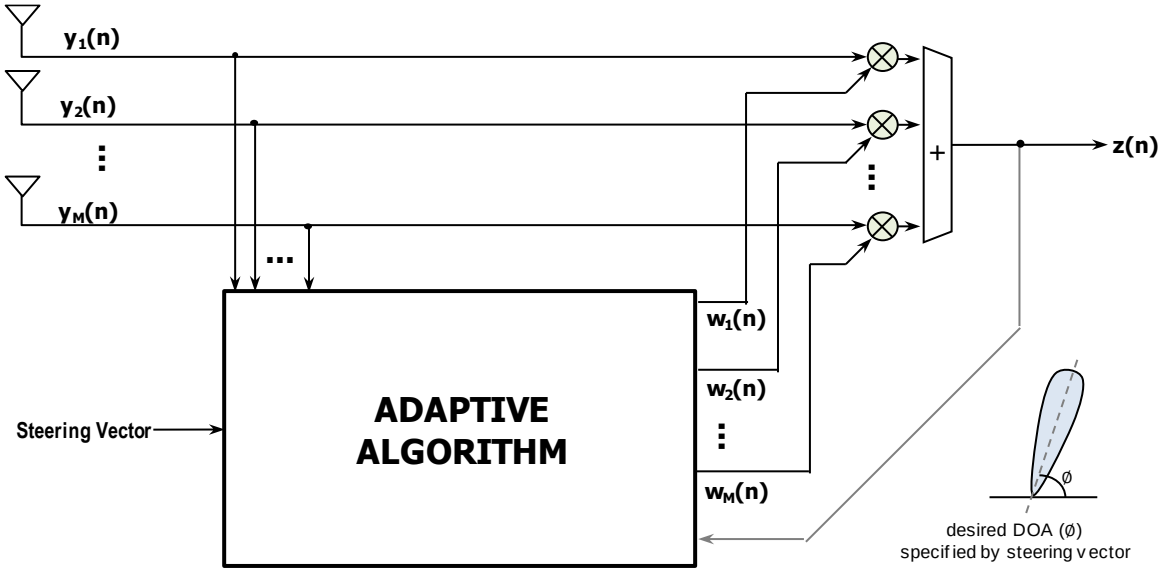

Figure C.1. Adaptive Beamforming Structure.

- For the beamformer, the output at a time n, $z(n)$ is given by a linear combination of the data at M sensors.
  - ✓ $\boldsymbol{y}(n)$: column vector of length M representing M complex input samples at time n.
  - ✓ $\boldsymbol{w}(n)$: column vector of length M containing the complex weights at time n.

$$z(n) = \boldsymbol{w}^H(n).\boldsymbol{y}(n)$$

INPUT SIGNALS

- For example, if we consider a linear array with M sensors, the samples at each sensor m ($m = 1,..M$) can be modeled as:

$$y_m[n] = s_m[n] + i_m[n] + r_m[n]$$

where $s_m[n]$ denotes the desired signal or signal-of-interest (SOI), $i_m[n]$ the interference (or jammer), and $r_m[n]$ the noise. Based on a baseband model, for a specific angle of arrival, each signal can be expressed as:

$$s_m[n] = s[n]e^{-j\vec{k}.\vec{x}_m}, \qquad i_m[n] = i[n]e^{-j\vec{k}.\vec{x}_m}$$

where $\vec{k}.\vec{x}_m = \pi \sin\phi\left(m - 1 - \left(\frac{M-1}{2}\right)\right)\frac{d}{\lambda/2}$ depends on the angle of arrival of the signal (be it $s[n]$ or $i[n]$).

TYPICAL BEAMFORMING ALGORITHMS

- **Least Mean Square (LMS)**:

$$z(n) = \boldsymbol{w}^H(n).\boldsymbol{y}(n)$$
$$\boldsymbol{w}(n+1) = \boldsymbol{w}(n) + \mu.\boldsymbol{y}(n).e^*(n),\ 0 < \mu < 1$$
$$e(n) = d(n) - z(n),\ d(n)\text{: desired response at time } n.$$

  - ✓ $d(n)$: desired signal coming from the sensors. This is a modeled response from the sensor in the desired direction of arrival. Signals coming from the sensors are noisy.
  - ✓ $\boldsymbol{w}(1) = [0\ 0\ ...\ 0]^T$
  - ✓ Parallelization: There is data dependency. However, the dot product between the weights and the input can also be implemented in parallel (*parallel_for* and *parallel_reduce*). In addition, the update equation for the weights (dot product and summation) resemble

SAXPY, except that we are now dealing with complex-valued data. $d(n)$ is constant for a desired direction of arrival. The user can change it at any time should a different direction of arrival is desired.

- **Frost (Constrained LMS)**:
  - ✓ For simplicity's sake, we use the notation $\boldsymbol{y}_n$ and $\boldsymbol{w}_n$ as opposed to $\boldsymbol{y}(n)$ and $\boldsymbol{w}(n)$.
  - ✓ N: number of snapshots (collection of M samples at time n). Also: $\boldsymbol{w}_1 = \boldsymbol{w}_C$.

$$\begin{aligned} &for\ n = 1{:}N \\ &\quad z(n) = \boldsymbol{w}_n^H . \boldsymbol{y}_n \\ &\quad \boldsymbol{w}_{n+1} = \boldsymbol{w}_C + P \times (\boldsymbol{w}_n - \mu z^*(n)\boldsymbol{y}_n) \\ &end \end{aligned}$$

where

$$P = I - C^H(CC^H)^{-1}C$$
$$\boldsymbol{w}_C = C^H(CC^H)^{-1}c$$

  - ✓ This adaptive algorithm solves a constrained optimization problem subject to $C\boldsymbol{w}_n = c$. We want $\boldsymbol{w}_n$ to accentuate signals coming from some direction ($\boldsymbol{y}_n^H . \boldsymbol{w}_n = 1$) and/or suppress jammers ($\boldsymbol{y}_n^H . \boldsymbol{w}_n = 0$). Thus, the quantities C and c can be used to steer the beamformer in a desired direction (i.e., switch to a desired beam pattern).
    - C: constraint matrix. It is formed from the set of ideal signals expressing various propagation directions.
    - c: column vector of constraining values. They are chosen to accentuate/select certain signals and suppress others.

  - ✓ We consider C to be formed from the set of steering vectors $a^H(\emptyset)$:

$$C = \begin{bmatrix} a^H(\emptyset_1) \\ a^H(\emptyset_2) \\ \vdots \\ a^H(\emptyset_M) \end{bmatrix}, a^H(\emptyset) = [e^{j\vec{k}.\vec{x}_1}\ e^{j\vec{k}.\vec{x}_2} \dots e^{j\vec{k}.\vec{x}_M}]$$

    - If, for example, we want to suppress every direction except for $\emptyset_1$, then $c = [1\ 0\ \dots\ 0]^T$. By judiciously selecting C and c, we can control the beam pattern of the antenna array.

  - ✓ Parallelization: There is data dependency. However, the dot product between the weights and the input can also be implemented in parallel (*parallel_for* and *parallel_reduce*). In addition, the update equation for the weights can be parallelized as well. Note that P and $\boldsymbol{w}_C$ are constant quantities for a desired propagation direction. The user can modify this at any time for a different propagation direction.

CYLINDER PRESSURE ESTIMATION

- Computing the instantaneous engine cylinder pressure in a spark ignition engine as a function of crank angle is extremely useful. It can be applied as a means to obtain the instant torque, the indicated mean effective pressure, and for estimating optimal ignition timing. However, due to the complexity of the combustion model, this information is not usually available.

- **Traditional approach**: Most modern spark ignition engines use a Look-Up Table (LUT):
  - ✓ The fuel, spark and valve timings are mapped as a function of engine speed (rpm), and load (manifold air pressure: MAP). The electronic engine control unit (ECU) then reads sensor inputs such as crank or cam encoders, manifold air pressure (MAP), mass air-flow rate and throttle position, and then commands controllers or actuators to produce the desired spark, fuel and valve timing.
  - ✓ While the LUTs can be quite large, especially with boosting and variable valve timing, the computing resources are still relatively modest. However, this traditional LUT approach does not scale well with changes in operating conditions and parameters, as the amount of required memory can grow very quickly.

- **Ideal approach**: Ideally, we would like to have physics-based combustion models that can run real-time in the ECU to reliably estimate cylinder pressure. At this point in time, the computing power required to do this is prohibitive.

- **Realistic approach**: A less computationally intensive approach is to generate a LUT of Wiebe functions from dynamometer data that predict the heat release rate, and then use this table along with a thermodynamics-based cylinder pressure model to calculate the real-time cylinder pressure. This approach can be used in Hardware-in-the-Loop (HIL) plant models or potentially implemented directly into the next generation engine ECUs. So, this pressure estimation model might be better suited to deliver crank-angle-resolved cylinder pressure in real-time.
- This discrete model for in-cylinder pressure estimation has been validated (via MATLAB® implementation) using 13 sets of data from two different engines and a range of speed/load conditions. It is also tuned for a range of engine conditions (speed, load, etc.) in order to generate accurate heat release rate and estimated cylinder pressure traces. This model is suitable for determining optimal spark timing.

DISCRETE MODEL FOR ENGINE CYLINDER PRESSURE ESTIMATION

- This pressure estimation model was derived using the 1st Law of Thermodynamics applied to the closed valve period of the engine cycle. As a result, the model only performs pressure estimation for the closed-valve portion of the engine cycle. This is the most computation-intensive portion of the cycle.
- The calculation starts at intake valve closing (IVC) and finishes at exhaust valve opening (EVO). Fig. D.1 shows the reference system for the crank angles as well as where IVC, EVO, and IGN (ignition time) are expected to occur. The complete engine cycle goes from -360° to 360° (1 cycle or 2 revolutions).

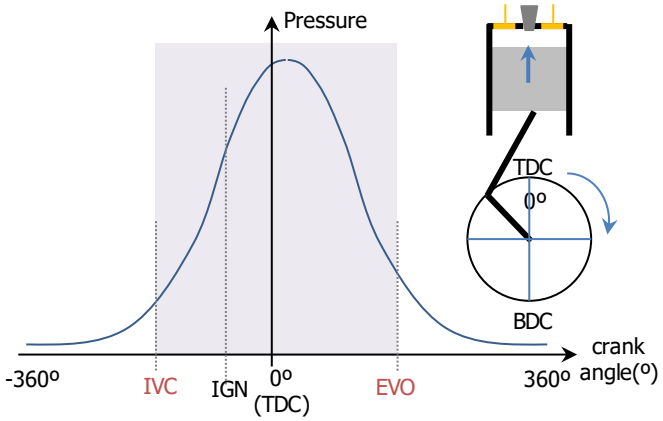

Figure D.1. Reference for the crank angles. Pressure and Heat Release are computed from IVC to EVO. TDC denotes Top dead-center, BDC denotes Bottom dead-center, and IGN denotes Ignition time (θ_0)

- The relationship among heat release rate, pressure, volume, and heat transfer rate is given by:

$$\frac{dQ_{HR}}{d\theta} = \frac{\gamma}{\gamma-1}P\frac{dV}{d\theta} + \frac{1}{\gamma-1}V\frac{dP}{d\theta} + \frac{dQ_{HT}}{d\theta}$$

DISCRETE MODEL

- It expresses the equation in terms of the discrete pressure per crank angle:

$$\left.\frac{dQ_{HR}}{d\theta}\right|_n = \frac{\gamma}{\gamma-1}P(n)\left.\frac{dV}{d\theta}\right|_n + \frac{1}{\gamma-1}V(n)\left(\frac{P(n+1)-P(n)}{\theta(n+1)-\theta(n)}\right) + \left.\frac{dQ_{HT}}{d\theta}\right|_n$$

  - ✓ $\theta(n)$: crank angle at time n.
  - ✓ $\theta(0)$ (or θ_0): Ignition Time.
- Pressure $P(n)$ then results:

$$P(n+1) = P(n) + \Delta\theta\left[\frac{(\gamma-1)}{V(n)}\frac{dQ_{HR}}{d\theta}\Big|_n - \gamma\frac{P(n)}{V(n)}\frac{dV}{d\theta}\Big|_n - \frac{(\gamma-1)}{V(n)}\frac{dQ_{HT}}{d\theta}\Big|_n\right]$$

- ✓ $\frac{dQ_{HT}}{d\theta}\Big|_n$: Heat Transfer rate from the cylinder gases to the cylinder walls.
- ✓ $\frac{dQ_{HR}}{d\theta}\Big|_n$: Heat Release rate.

Heat Transfer Rate

- There is an empirical model for the heat lost through the walls of the chamber. It is based on the heat transfer coefficient (h_{corr}) adapted from the Woschni equation.

$$\frac{dQ_{HT}}{d\theta}\Big|_n = \frac{dQ_{HT}}{dt}\frac{dt}{d\theta} = h_{corr}(n)A_{ch}(n)\big(T_g(n) - T_w\big)\frac{30}{N\pi}, N = rpm$$

$$h_{corr}(n) = c \times 0.013 \times V(n)^{-0.06} \times P(n)^{0.8} \times T_g(n)^{-0.4} \times \big(\overline{v_p} + 1.4\big)^{0.8}$$

$$V(n) = V_C\left[1 + \frac{1}{2}(r_c - 1)\left[\hat{l} + 1 - cos\theta_n - \big(\hat{l}^2 - sin^2\theta_n\big)^{\frac{1}{2}}\right]\right]$$

- ✓ $\overline{v_p}$: mean piston speed.
- ✓ $V(n)$: chamber volume per crank angle.
  - r_c: compression ratio
  - V_C: clearance volume.
  - $\hat{l}$: ratio of the connecting rod length to the crank throw.
  - The derivative of the volume with respect to the crank angle can then be obtained. Here $V_D = \frac{1}{2}V_C(r_c - 1)$ is called the displacement volume.

$$\frac{dV}{d\theta}\Big|_n = \frac{1}{2}V_D sin\theta_n\left[1 + cos\theta_n/\big(\hat{l}^2 - sin^2\theta_n\big)^{\frac{1}{2}}\right]$$

- ✓ T_w: temperature of the cylinder wall.
- ✓ $T_g(n)$: Gas temperature in the cylinder. It results from the ideal gas law: $T_g(n) = \frac{P(n)\times V(n)}{MR}$
  - R: universal gas constant
  - M: mass of the trapped gas: $M = \frac{AF+1}{1-r_f} \times m_f$
    - r_f: residual fraction
    - AF: Air/fuel ratio
    - m_f: mass of fuel in the cylinder
- ✓ γ: specific heat ratio. It can be computed as: $\gamma = 1.392 - 8.13 \div\times 10^{-5} \times T_g(n)$
- ✓ $A_{ch}(n)$: surface area at every crank angle: $A_{ch}(n) = A_{ch}(TDC) + \frac{\pi}{2}BS\left[\hat{l} + 1 - cos\theta_n - \big(\hat{l}^2 - sin^2\theta_n\big)^{\frac{1}{2}}\right]$
  - $A_{ch}(TDC)$: surface area of the chamber at top dead center (TDC)
  - B: Bore
  - S: Stroke

Heat Release Rate

- A model for the heat release rate is presented here. The formula starts working at spark timing (θ_0) and finishes after burn duration ($\Delta\theta_B$).
  - ✓ η_c: combustion efficiency.
  - ✓ m_f: mass of the fuel in the cylinder.
  - ✓ LHV: lower heating value.
  - ✓ α and β: parameters used to calibrate the model.
  - ✓ The mass fraction burned ($\frac{Q_{HR}}{\eta_c m_f LHV}$) can be modeled using the Wiebe function. The Heat Release rate can then be expressed in terms of the Wiebe function:

$$\frac{dQ_{HR}}{d\theta}\Big|_n = \begin{cases} 0, & \theta_n \le \theta_0 \\ \dfrac{\eta_c m_f LHV\alpha(\beta+1)}{\Delta\theta_B(1-e^{-\alpha})}\left(\dfrac{\theta_n-\theta_0}{\Delta\theta_B}\right)^{\beta} \times e^{\left[-\alpha\left(\frac{\theta_n-\theta_0}{\Delta\theta_B}\right)^{\beta+1}\right]}, & \theta_n > \theta_0 \end{cases}$$

MODEL CALIBRATION

- The heat release and heat transfer models are calibrated based on actual pressure traces. The model only needs to store the heat transfer and heat release parameters in order to generate an estimated pressure trace.

- Most Heat Transfer model parameters are directly obtained by the engine data and the operating conditions (rpm, chamber pressure, load, etc.)

- The calibration procedure for a particular engine and operating conditions might go as follows (this results in the parameters α, β, c, and $\Delta\theta_B$)
  1. Given the actual pressure trace, pick an initial value of c in order to complete a tentative heat transfer rate model. A tentative heat release rate is then computed.
  2. Plot the cumulative heat release rate (Q_{HR}). The maximum value is the total heat release, whose value should equal the actual fuel energy released, $\eta_c m_f LHV$. If the cumulative heat release does not match the fuel energy release, then adjust c, so that the cumulative heat release reaches $\eta_c m_f LHV$. Alternatively, the mass fraction can be plotted versus crank angle, where the mass fraction is the cumulative heat release divided by $\eta_c m_f LHV$. Then, adjust c until the mass fraction reaches 1. This is a preferred approach.
  3. At this point, the actual heat release rate and the heat transfer rate models are calibrated. With the actual mass fraction, one can get the burn duration ($\Delta\theta_B$), counted here from 0 to 97% of the mass fraction (it should be noted that some researchers use 10-90% or other variations that affect the Wiebe constants).
  4. A non-linear curve-fitting code can be used to fit the Wiebe function to the cumulative heat release curve. This step will result in α and β that are parameters of the heat release rate model.
  5. With the complete models for the heat transfer rate and heat release rate, we can now compute the estimated pressure trace per crank angle.

- We can store these parameters resulting from calibration for different engines and operating conditions. They can be loaded at run-time. It is much more efficient to store the calibration parameters rather than the actual pressure traces.

- Parallelization: Once the calibration parameters are known, the different components of the pressure model can be computed in parallel. Moreover, each component consists of multiple operations that allow us to explore different strategies as the computations are unbalanced.

- The process can be extended to cover the full operating range of an engine. Cylinder pressure data can be collected that covers the entire operating range of a given engine. For a set of specific conditions (e.g.: load, rpm, valve timing), the heat release and heat transfer parameters can be determined. The parameter space can be interpolated to cover the full operating space and then used to produce sets of heat release parameters that cover the full range of speed, load, spark timing, etc.