



ARTIFICIAL NEURAL NETWORKS AND DEEP LEARNING (PART III)

CS576 MACHINE LEARNING



Instructor: Dr. Jin S. Yoo, Professor
Department of Computer Science
Purdue University Fort Wayne

Reference

- Kelleher et al., Fundamentals of Machine Learning (2nd edition), Ch 8



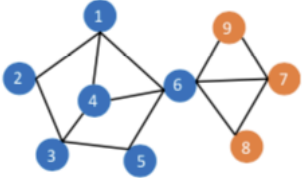
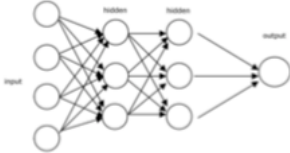
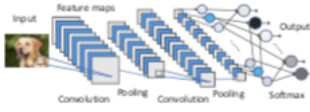
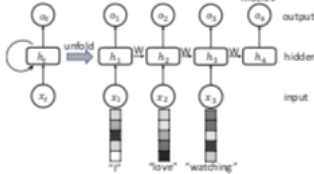
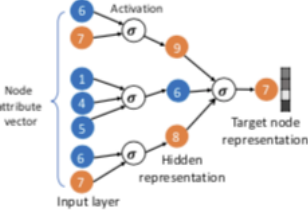
Part III Outline

- Extensions and variations (cont.)
 - **Convolutional Neural Networks**
 - Introduction to Convolutional Neural Networks
 - Weight Sharing and Translation Equivalent Feature Detection
 - Filter hyper-parameters: Dimensions, stride and padding
 - Pooling
 - Training CNNs – Weight Update

From Neural Networks to Deep Learning

- **Deep Learning** refers to training (deep) neural networks with many layers
 - More neurons, more layers
 - More complex ways to connect layers
- Deep Learning has some major advantages, making it popular
 - Tremendous improvement of performance in many tasks
 - Image recognition, natural language processing, AI game playing...
 - Requires no (or less) feature engineering, making end-to-end models possible
- Several factors lead to deep learning's success
 - Very large data sets
 - Massive amounts of computation power (GPU acceleration)
 - Advanced neural network structures and tricks
 - Convolutional neural networks, recurrent neural networks, graph convolution network ...
 - Dropout, ReLU, residual connection, ...

Overview of Deep Learning Architecture

Data type 	Multi-dimensional Features: credit rating, account balance $x = (4.5, 500, 3, 5)$ #deposits, #withdraws	Grid 	Sequence $x = \text{"I love watching movies."}$	Graph 
DL Architecture	Feed-forward Network 	CNN 	RNN 	GNN 

- **Feed-forward neural networks** for *multidimensional* data
 - where each data tuple is represented as an m -dimensional vector for attributes $X = (x_1, x_2, \dots, x_m)$ where m is the number of attributes
- **Convolutional neural networks (CNNs)** for *grid-like* data
- **Recurrent neural networks (RNNs)** for *sequence* data
- **Graph neural networks (GNNs)** for *relational* (i.e., graph) data.

Convolutional Neural Networks

- One of the most successful deep learning models is **convolutional neural networks (CNNs)**.
- CNNs have been successfully applied for various applications:
 - Various **image object recognition tasks** - face recognition in the challenging scenario (e.g., upside down)
 - Healthcare – detection of biomarkers, access of patient's health risks and drug discovery
 - Applications to **time series** data - forecasting the feature measurements and detection of anomalies
 - Computer game – AlphaGo

Convolutional Neural Networks

- CNNs are designed for *grid-like data*, such as images.
- Mathematically, the input grid data for CNNs are represented as a *multidimensional array* (i.e., a *tensor*).
- Examples:
 - The time-series data can be represented as a single-dimensional array (a 1-D tensor) whose elements provide the measures at the corresponding time stamps
 - The gray image can be represented as a matrix (a 2-D tensor) whose elements measure the gray scales of the corresponding pixels
 - The color-image can be represented as a 3-D tensor with third mode representing different color channels (e.g., red, green, blue)

Example: MNIST Dataset for CNN

- The CNN architecture was originally applied to handwritten digit recognition
- Much of the early work on CNNs was based on the MNIST (pronounced *em-nist*) dataset(Le Cun et al., 1998).
 - The dataset contains 60,000 training and 10,000 test images of handwritten digits from approximately 250 writers.
 - Each image is labeled with the digit it contains.
 - Each image is grayscale and can be represented as a grid of 28 by 28 integers in the range [0, 255] where a 0 value indicates a white pixel, a value of 255 indicates a black pixel, and numbers between 0 and 255 indicate shades of gray.
 - The prediction task is to return the correct label for each image.

MNIST Data



Figure : Samples of the handwritten digit images from the MNIST dataset.

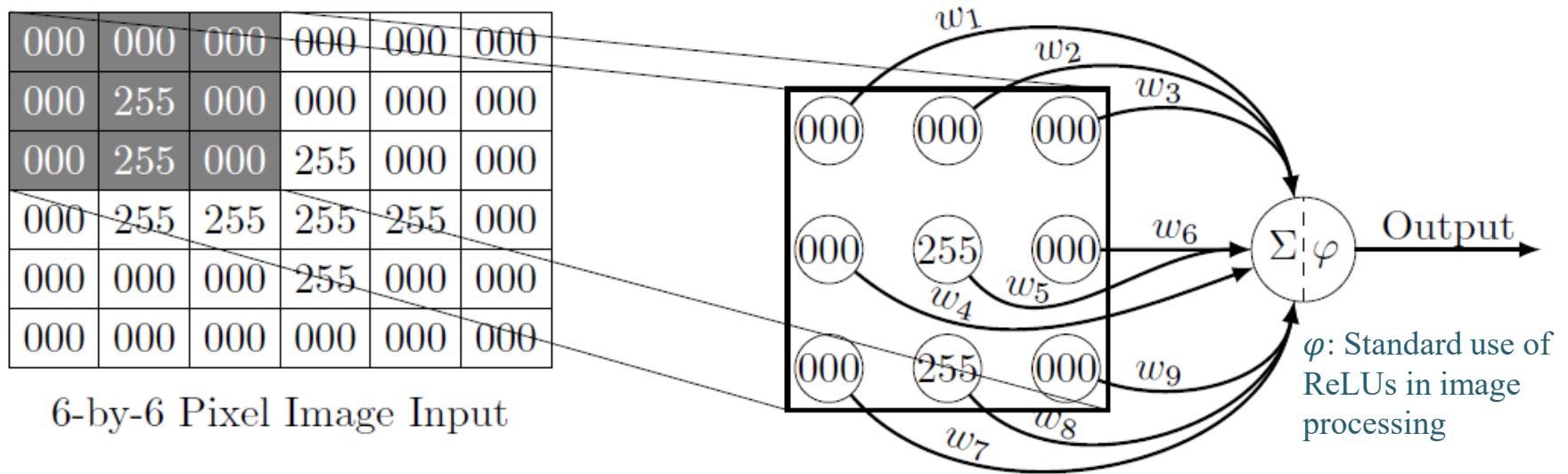
CNN's Characteristics

- Convolutional neural networks have three distinctive characteristics:
 1. local receptive fields;
 2. weight sharing; and
 3. sub-sampling (pooling).

Local Receptive Fields

- In neural networks, neurons are organized to focus on specific small areas of the visual field, each looking for single features.
- These areas are referred to as **local receptive fields**.
- This arrangement simplifies the learning task for each neuron, which needs to determine the presence of specific features within its small, designated region, as opposed to recognizing features dispersed throughout the broader visual field.
- This setup enables neurons to become specialized in recognizing basic elements of the input, like lines or edges in an image. These basic elements are then passed up to subsequent layers, where neurons combine them to identify more complex features.

Example: Local Receptive Field in CNNs



**** The neuron receives inputs only from a small predefined region of the input, i.e., only from the pixels in its **local receptive field**.**

Figure: The input data of a 6-by-6 matrix that represents a grayscale image where the number ‘4’ is depicted through pixel values with 255, and a neuron with a **local receptive field** that covers the top-left corner of the image.

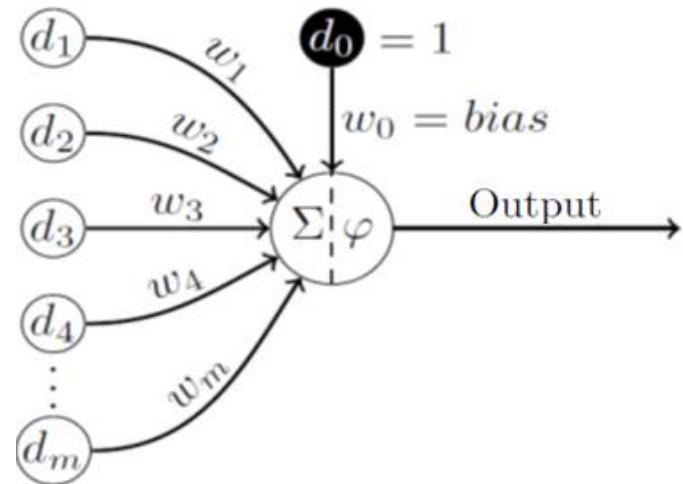
- This neuron focuses on the top-left 3x3 segment of the image, processing its 9 pixels in a grid format that reflects the image's two-dimensional structure.
- Beyond the grid-like input configuration, this neuron processes information similarly to the way neurons function in feed-forward neural networks.

Comparison with A General Artificial Neuron

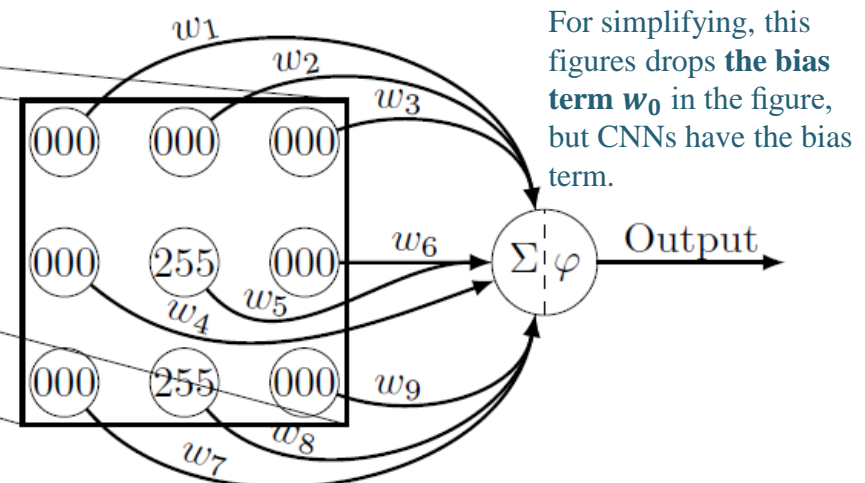
- An artificial neuron for general neural network
 - An input example, $d = \langle d_1, \dots, d_m \rangle$
- An artificial neuron for convolutional neural network
 - An input image example

000	000	000	000	000	000
000	255	000	000	000	000
000	255	000	255	000	000
000	255	255	255	255	000
000	000	000	255	000	000
000	000	000	000	000	000

6-by-6 Pixel Image Input



(a) A schematic of a neuron for general neural networks

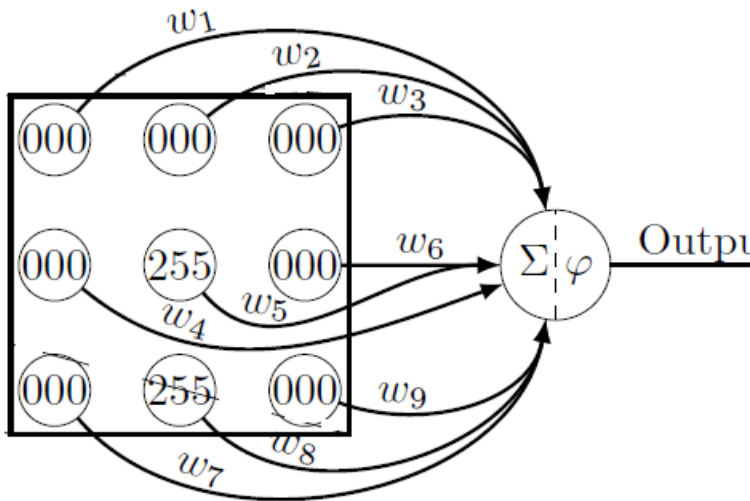


(b) A schematic of a neuron for convolutional neural networks

Example - Weights and Activations

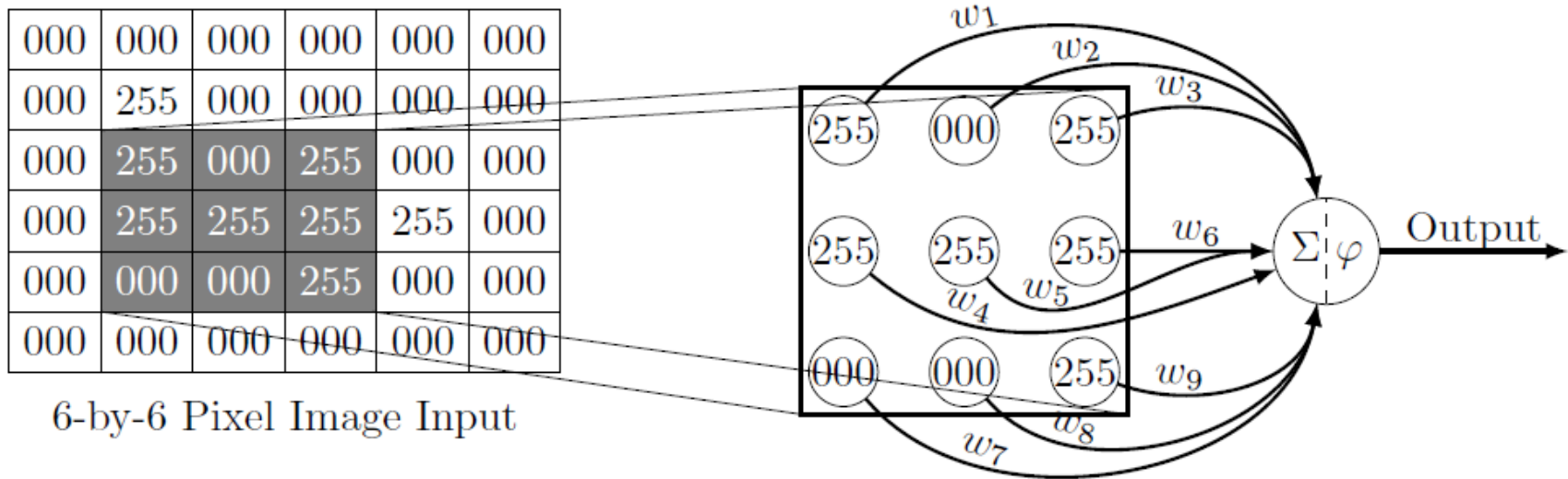
- Assume a set of **weights** for this neuron is : $\begin{bmatrix} 0 & 0 & 0 \\ 1 & 1 & 1 \\ 0 & 0 & 0 \end{bmatrix}$
 - This matrix of weights is designed to detect a horizontal line in the middle of the neuron's receptive field.

- Activation: $a_i = \text{rectifier}(\sum_{j=1}^9 w_j \times \text{input}_j)$



$$\begin{aligned}
 a_i &= \text{rectifier}((w_1 \times 000) + (w_2 \times 000) + (w_3 \times 000) \\
 &\quad + (w_4 \times 000) + (w_5 \times 255) + (w_6 \times 000) \\
 &\quad + (w_7 \times 000) + (w_8 \times 255) + (w_9 \times 000)) \\
 &= \text{rectifier}((0 \times 000) + (0 \times 000) + (0 \times 000) \\
 &\quad + (1 \times 000) + (1 \times 255) + (1 \times 000) \\
 &\quad + (0 \times 000) + (0 \times 255) + (0 \times 000)) \\
 &= 255
 \end{aligned}$$

Example - Receptive Field and Activation



The same weight set is

$$\begin{bmatrix} 0 & 0 & 0 \\ 1 & 1 & 1 \\ 0 & 0 & 0 \end{bmatrix}$$

$$\begin{aligned} a_i &= \text{rectifier}((w_1 \times 255) + (w_2 \times 000) + (w_3 \times 255) \\ &\quad + (w_4 \times 255) + (w_5 \times 255) + (w_6 \times 255) \\ &\quad + (w_7 \times 000) + (w_8 \times 000) + (w_9 \times 255)) \\ &= \text{rectifier}((0 \times 255) + (0 \times 000) + (0 \times 255) \\ &\quad + (1 \times 255) + (1 \times 255) + (1 \times 255) \\ &\quad + (0 \times 000) + (0 \times 000) + (0 \times 255)) \end{aligned}$$

$$= 765$$

With the weight set, a maximum activation when horizontal line is present across the middle row of the receptive field.

Variations in Weight Matrices

- Weight matrices dictate the activation patterns of neurons in response to various inputs.
- The extensive range of possible weight combinations allows for the detection of a diverse range of features.
- Example weight matrices.

$$\begin{bmatrix} 0 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \end{bmatrix}$$

(a)

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

(b)

$$\begin{bmatrix} -1 & +1 & -1 \\ -1 & +1 & -1 \\ -1 & +1 & -1 \end{bmatrix}$$

(c)

$$\begin{bmatrix} -1 & -1 & -1 \\ +1 & +1 & +1 \\ -1 & -1 & -1 \end{bmatrix}$$

(d)

Only integer weights are used in examples for clarity.

Variations in Weight Matrices (cont.)

■ Edge Detection

- Example:

$$\begin{bmatrix} 0 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \end{bmatrix} \quad \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

(a)

(b)

- Weight matrix (a): Activates for vertical edges in the center column.
- Weight matrix (b): Activates for diagonal edges from left to right.

■ Contrast Detection

- Weight matrices with positive and negative values can detect contrast.
- Neurons with these matrices respond to more complex visual patterns.

- Example:

$$\begin{bmatrix} -1 & +1 & -1 \\ -1 & +1 & -1 \\ -1 & +1 & -1 \end{bmatrix} \quad \begin{bmatrix} -1 & -1 & -1 \\ +1 & +1 & +1 \\ -1 & -1 & -1 \end{bmatrix}$$

(c)

(d)

- Weight matrix (c) : Activates for pixels with high values in the central column of the receptive field than there are in the other pixels in the input.
- Weight matrix (d) : Activates for pixels with high values in the central row of the receptive field.

Filters and Learning

- In the context of CNNs, **weight matrices** are referred to as **filters**.
- **Filters (Weights)** are learned during training, allowing the network to adapt to recognize useful visual patterns.
- Learning is a process of adjusting filter weights to improve task performance.

Outline

- Convolutional Neural Networks
 - Introduction to Convolutional Neural Networks
 - ☞ **Weight Sharing and Translation Equivalent Feature Detection**
 - Filter hyper-parameters: Dimensions, stride and padding
 - Pooling
 - Training CNNs – Weight Update

Translationally Equivariant

- A neuron acts as a detector for local visual features by applying a filter to its receptive field, which is essentially a specific pattern of input values.
- However, an effective image processing system is designed to recognize the presence of a visual feature anywhere in the image.
- This capability of a model to identify features regardless of their position in the image is technically referred to as being **translationally equivariant**.

Weight Sharing

- CNNs facilitate **translationally equivariant feature detection** via the mechanism of **weight sharing**.
- The weight sharing mechanism involves **grouping neurons** such that each neuron within the group uses an identical filter (weights) to process their respective inputs.
 - Consequently, when a group of neurons employs a common filter, each individual weight within the filter is repeatedly applied during the training algorithm's forward pass, processing the input multiple times, corresponding to the number of neurons utilizing that filter.
- This architecture allows the network to detect a feature across the entire input space.

Organization of Local Receptive Fields

- CNNs arrange the local receptive fields of neurons that utilize a common filter (thereby sharing the weights) in such a way that:
 - 1) the receptive field of each neuron **overlaps with adjacent regions**, ensuring a unique portion of the visual field is covered by each neuron; and
 - 2) the collective receptive fields of these neurons **cover the entirety of the visual field** without missing.

Neurons Sharing Weights and Their Local Receptive Fields

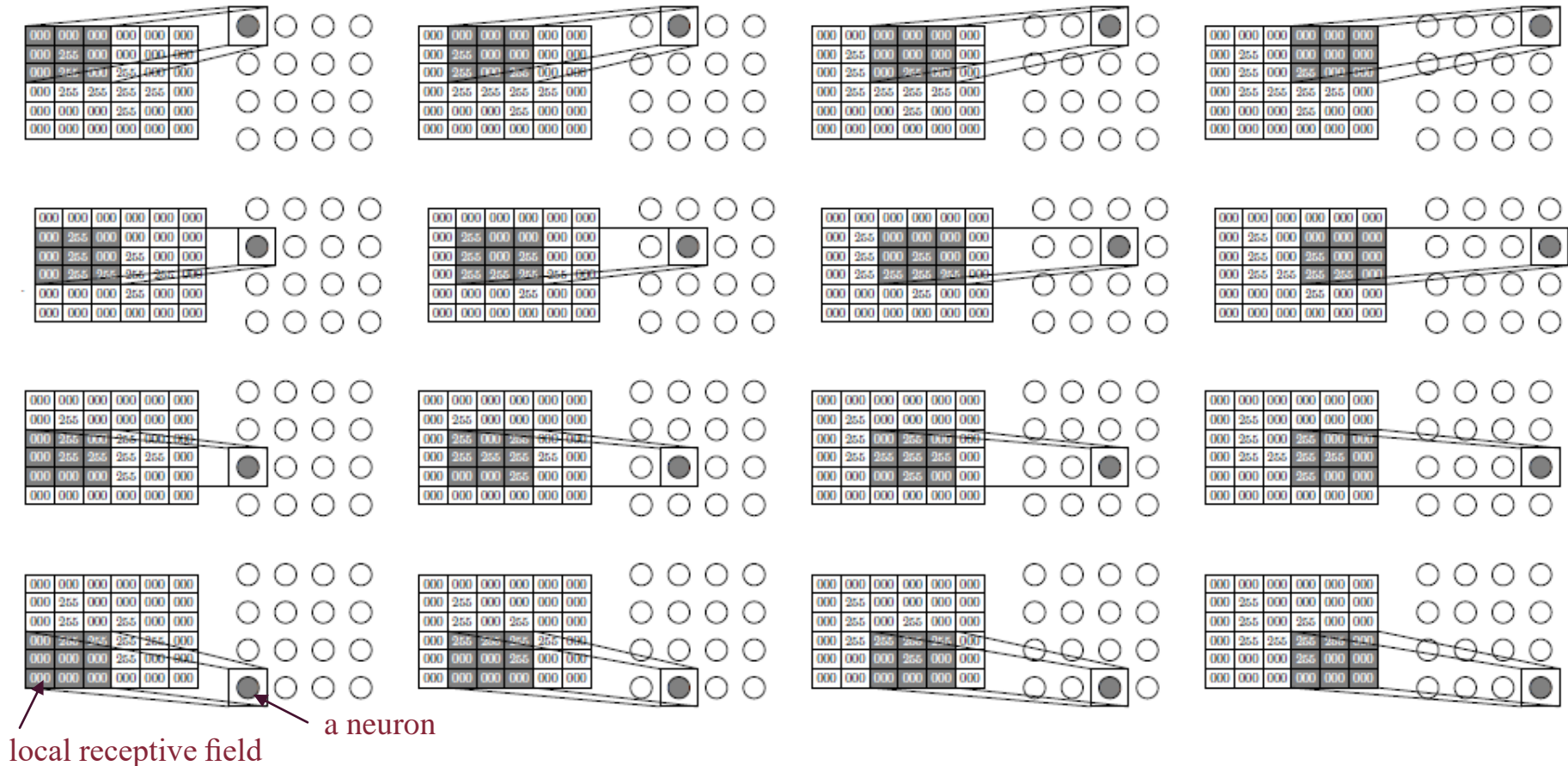


Figure. Illustration of the configuration of a group of neurons sharing weights (utilizing the identical filter) and their corresponding local receptive fields arranged to collectively encompass the entire input image.

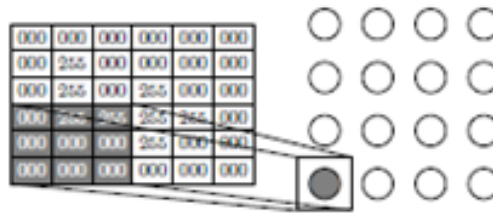
“Convolution” Term

- The term **convolutional neural network** is derived from a specific mathematical operation known as **convolution**, which involves applying a **filter** (or **kernel**) to the data.
- In this context, a convolutional neural network utilizes a collection of neurons that share a common filter.
- This filter is applied consecutively to different regions of the input image by a neuron, which processes the information and retains the outcome for each specific region.

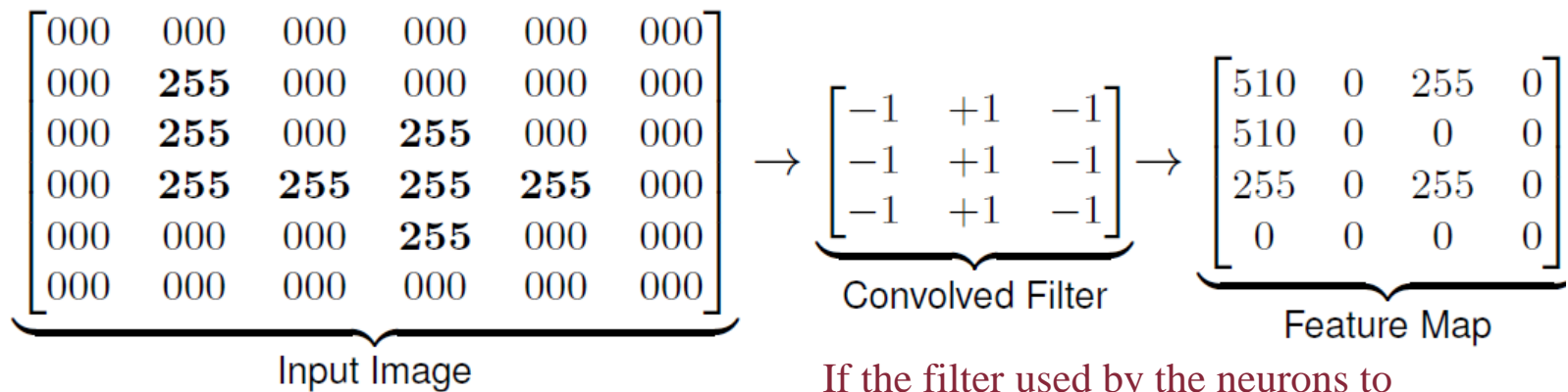
Feature Maps

- The collective set of neuron activations that corresponds to a shared filter is known as a **feature map**, highlighting the specific areas activated by the feature of interest.
- Feature maps are generated by applying a filter to the input and recording the activations.

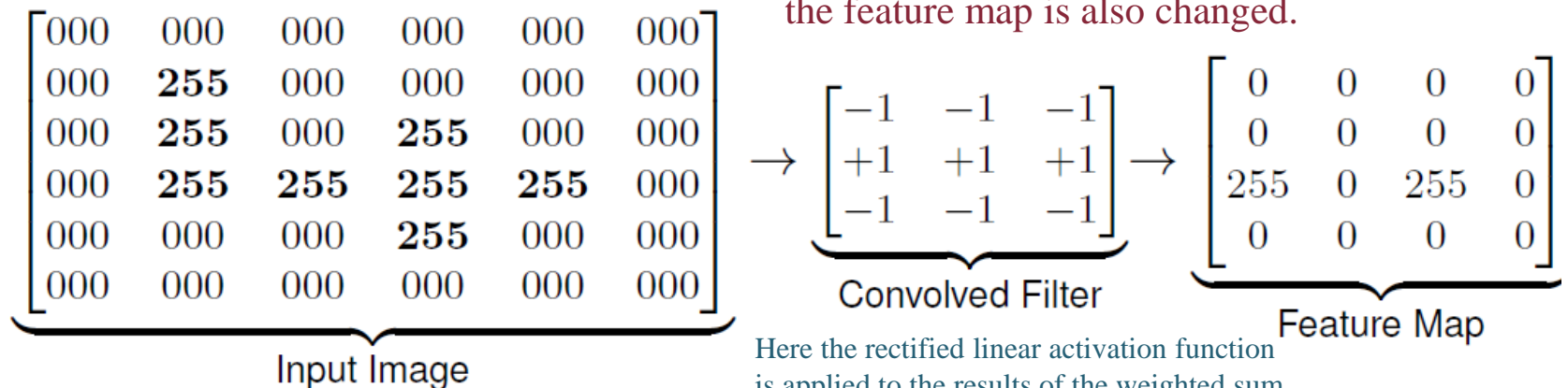
Filters and Generated Feature Maps



Example 1:



Example 2:



If the filter used by the neurons to process the example input is changed, the feature map is also changed.

Here the rectified linear activation function is applied to the results of the weighted sum of each receptive field and the filter.

Outline

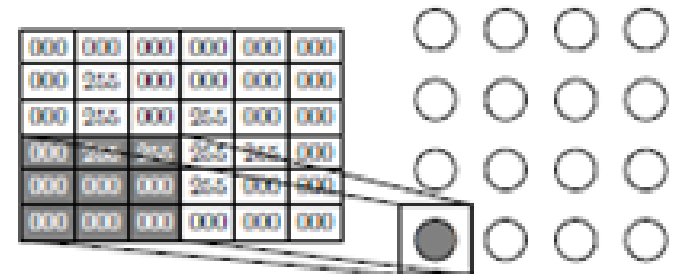
- Convolutional Neural Networks
 - Introduction to Convolutional Neural Networks
 - Weight Sharing and Translation Equivalent Feature Detection
 - 👉 **Filter hyper-parameters: Dimensions, stride and padding**
 - Pooling
 - Training CNNs – Weight Update

Hyper-parameters

- The **size of a feature map**, created when a filter is applied to an input, is defined by the number of neurons engaged in processing the input, where **each feature map element** represents the output from a single neuron.
- The number of neurons needed to analyze the full extent of an input, thereby shaping the size of the produced feature map, is influenced by three hyperparameters: the **dimensions of the filter, the stride, and the padding.**

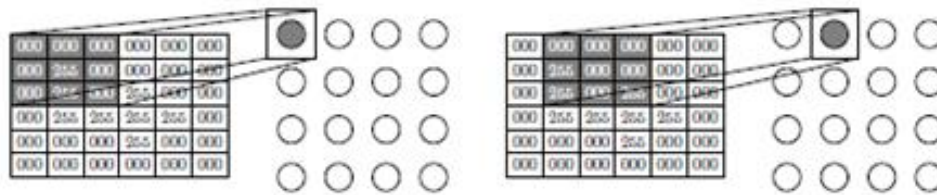
Filter Dimension Parameter

- The **filter dimension** hyper-parameter determines the **filter's size** along each dimension.
- The choice between larger or smaller filters **impacts the number of neurons** needed to span the input space—a larger filter means fewer neurons are required, and the inverse is true for smaller filters.
- For example, a 3-by-3 filter dimension
 - Reducing the filter to a 2-by-2 size necessitates expanding the neuron layer to a 5-by-5 grid to achieve complete input coverage, resulting in a larger 5-by-5 feature map.
 - Conversely, increasing our filter to a 4-by-4 size allows the input to be covered by a more compact 3-by-3 neuron layer, yielding a smaller 3-by-3 feature map.
- Selecting the most effective filter size for a specific dataset typically requires a process of trial and error, testing various sizes to determine the optimal configuration.



Stride Parameter

- The **stride** parameter determines the spacing between the centers of adjacent neurons' local receptive field within the grid of neurons utilizing the same filter.
- Example: Utilizing both a horizontal and vertical stride of 1



- A stride of 1 for both dimensions indicates significant overlap among the receptive fields of neighboring neurons.
- It's feasible to implement various strides in the horizontal and vertical directions.
 - E.g., Applying a stride of 3 would eliminate overlap between neighboring neurons' receptive fields and decrease the total number of neurons needed to span the entire input area.
- Selecting an optimal stride for a specific dataset typically requires a process of trial and error.

Motivations of Padding

- **Observation 1:** Utilizing a 4-by-4 grid of neurons to process a 6-by-6 input matrix results in a feature map with dimensions of 4-by-4. To prevent losing information through reduced dimensionality from input to feature map, alternative strategies may be required.
- **Observations 2:** The frequency at which each pixel is utilized as an input by the neuron grid varies.
 - The greatest disparity is seen when comparing the usage of edge pixels, like those at the corners of the image, to those at the center.
 - E.g., The top-left pixel is included in only one receptive field, while the pixel at the center position (3, 3) is encompassed by nine receptive fields.
- These outcomes (Observations 1 and 2) arise from the methodology of applying the filter solely to the **valid** pixel locations in the image.

Padding

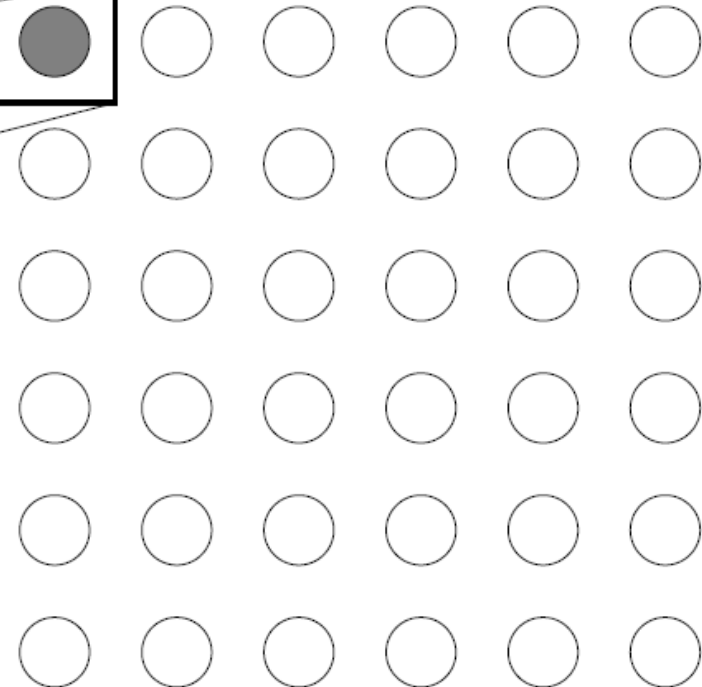
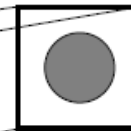
- The periphery of an image can be extended with **padding pixels**.
 - The term ‘**valid**’ is employed to differentiate between the actual pixels present in the image and the additional padding pixels that may be added around the image's edge.
- Advantages of padding
 - While there is still a variation in how often certain valid pixels are engaged by neurons, the application of padding has mitigated this difference of the size of input image and generated feature map.
- Similar to deciding on filter size and stride length, the choice to apply padding depends on the specific requirements of the task and typically involves a process of trial and error to optimize.

Example: Padding

A two-dimensional
3- by-3 filter

000	000	000	000	000	000	000	000
000	000	000	000	000	000	000	000
000	000	255	000	000	000	000	000
000	000	255	000	255	000	000	000
000	000	255	255	255	255	000	000
000	000	000	000	255	000	000	000
000	000	000	000	000	000	000	000
000	000	000	000	000	000	000	000

valid (real) pixels



6-by-6 Pixel Image Input
with 2 Rows and 2 Columns of Padding

imaginary pixels (with a value of 000)

Convolution of a Filter via

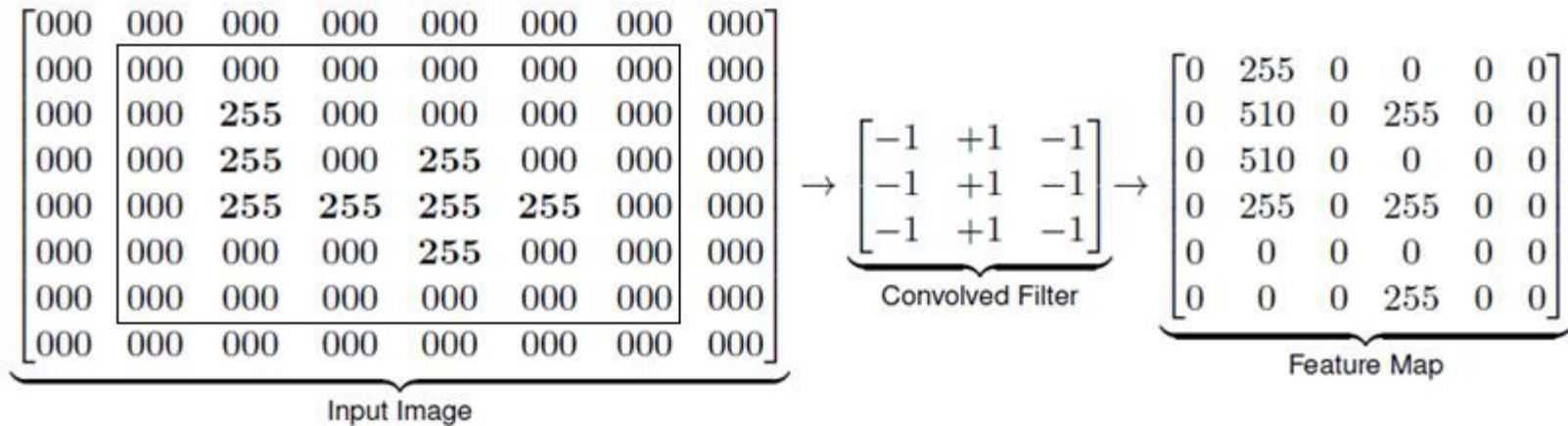
Set of Neurons Sharing Weights

* Each valid pixel at the corners is included in four receptive fields instead of just one, as was the case before padding. The number of receptive fields including a central pixel remains unchanged by padding. Adding this padding to the image increases the number of neurons required to cover the image

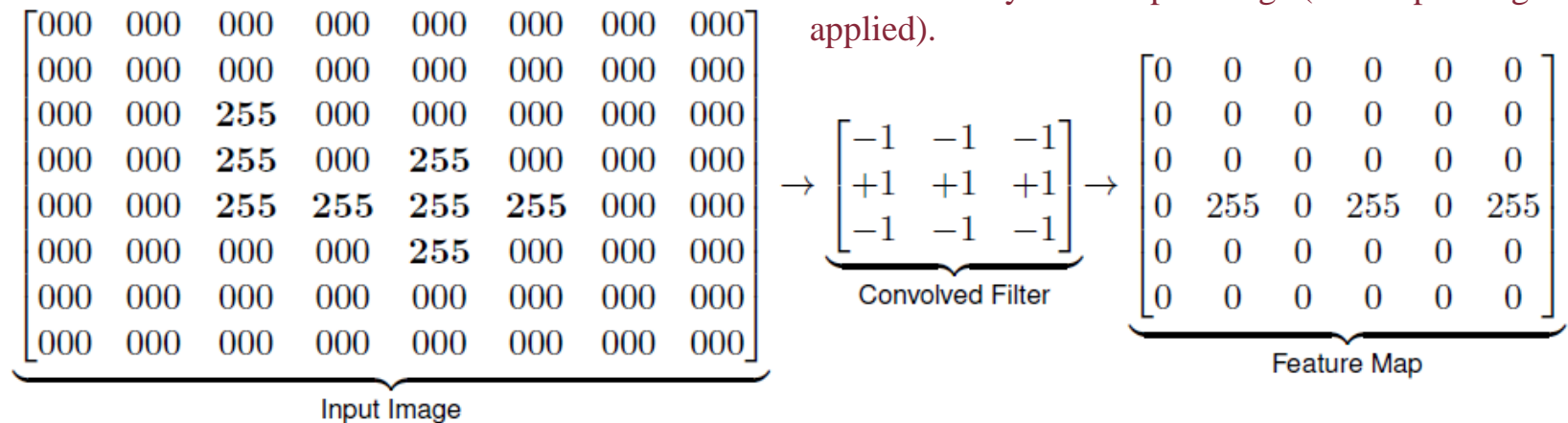
Figure An image of a number 4 after padding (the imaginary pixels shown in gray) has been applied to the original 6-by-6 matrix representation, and the local receptive field of a neuron that includes both valid (real) and padded pixels.

Generated Feature Maps with Padding

Example 1:



Example 2:

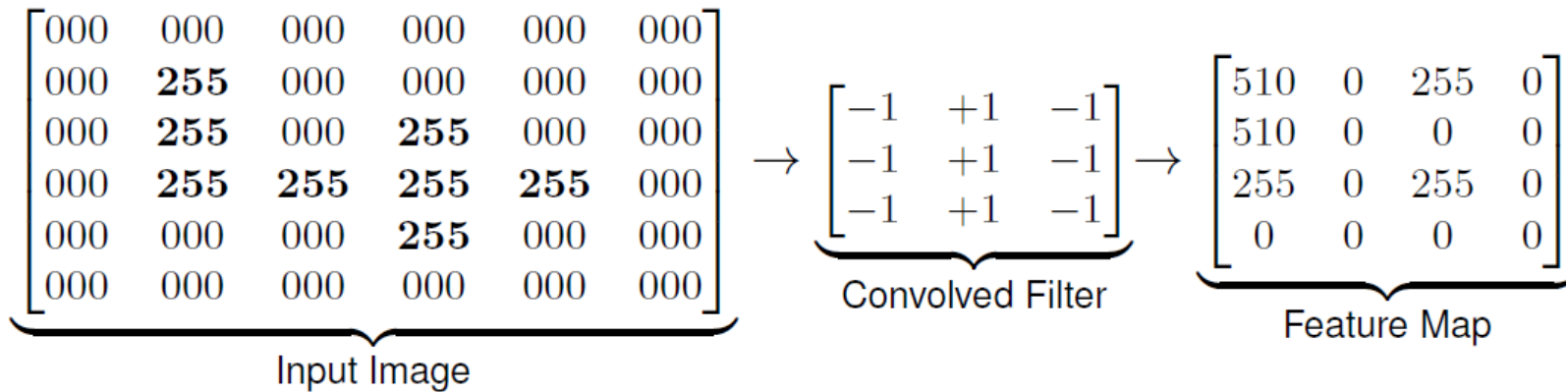


Now, the generated feature maps have the same 6-by-6 dimensionality as the input image (before padding was applied).

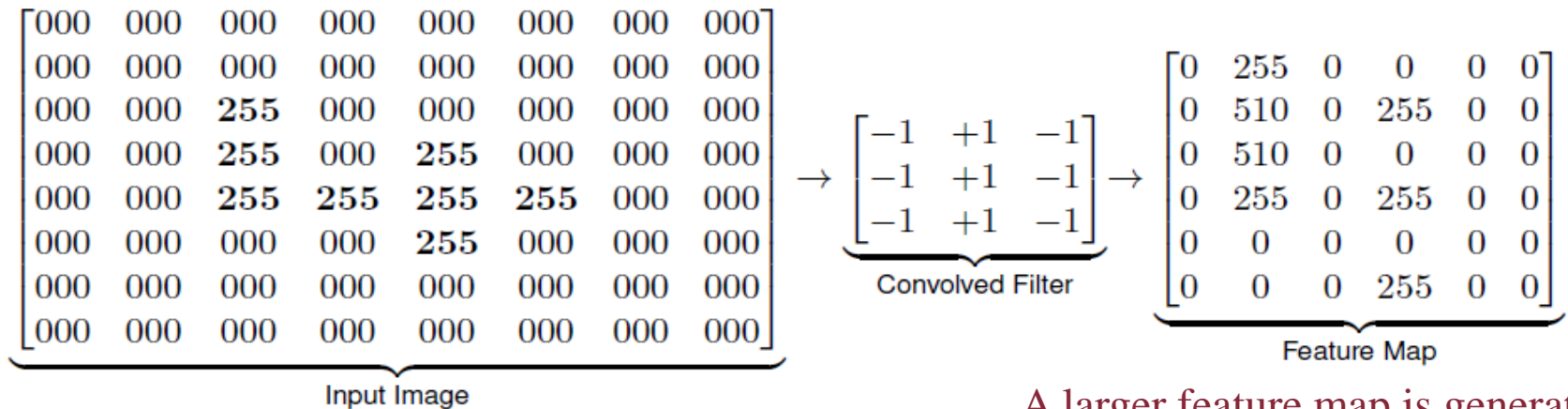
Figure: After the input has had padding applied and using a stride length of 1, a filter weight matrix and the feature map generated by using a set of neurons to process the image example input, 4.

Examples with/without Padding

Without padding



With padding



A larger feature map is generated.

Summary of Hyper-Parameters in CNNs

- The selection of filter size, stride length, and padding is determined by the specific requirements of the task at hand.
- It is common practice to choose a stride length of 1 and augment the image with padding, as suggested by Charniak (2019).
 - This approach guarantees that the dimensionality of the output from a layer of neurons, which processes the image with a filter, matches that of the input image.
- Ensuring consistent dimensionality between the input and output is particularly crucial in convolutional neural networks utilizing multiple neuron layers.
 - Here, the output from one layer serves as the input for the following layer. Without using padding to preserve dimensionality, the size of the input to each successive layer would progressively decrease.

Outline

- Convolutional Neural Networks
 - Introduction to Convolutional Neural Networks
 - Weight Sharing and Translation Equivalent Feature Detection
 - Filter hyper-parameters: Dimensions, stride and padding
 - 👉 **Pooling**
 - Training CNNs

Pooling

- In many image-processing tasks, the exact position of a visual feature within an image is not crucial.
 - E.g., Recognizing the presence of an eye in the upper left quadrant may be sufficient for face recognition without needing to pinpoint its exact pixel location, such as pixel (998,742).
- During the training phase of an image processing model, the **goal** is to achieve generalization beyond the exact spatial positions of features found in training datasets, enabling the model to detect these features even when they appear in new, varied positions in different images.
- To abstract from the precise locations of visual features, a common technique is to **sub-sample the feature maps**, thereby reducing the spatial resolution and focusing on higher-level attributes.

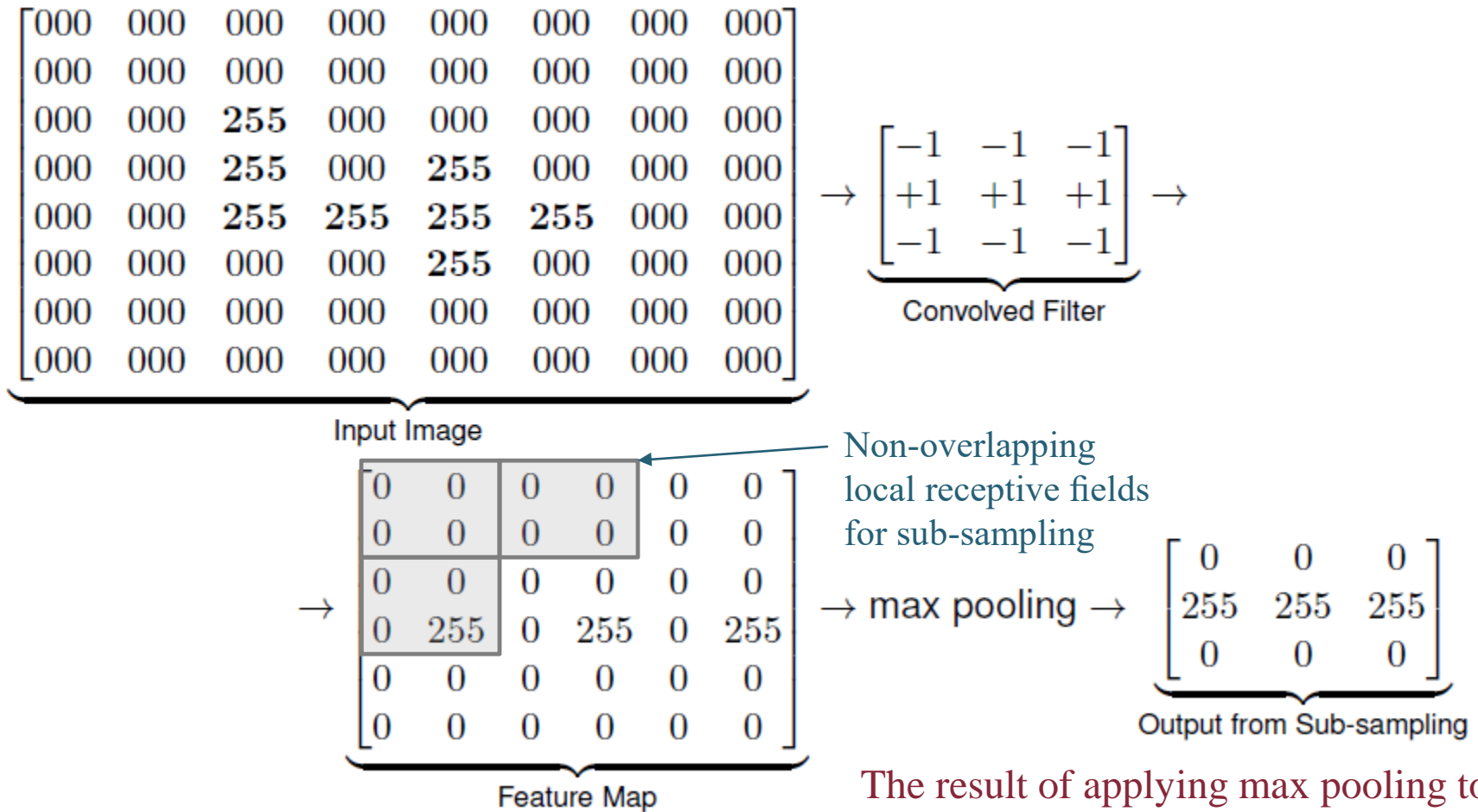
Pooling – Sub-Sampling Layers

- In CNNs, sub-sampling (dimensionality reduction) is often achieved through the use of **sub-sampling layers**.
 - Each neuron within a sub-sampling layer is associated with a distinct region of the feature map created by the previous layer, which itself is the result of applying a convolutional filter to the input.
 - Unlike in convolutional layers where receptive fields can overlap, **in sub-sampling layers, the receptive fields typically do not overlap**. This means each neuron's receptive field is unique to it within the layer.
 - This lack of overlap **results in fewer overall output activations from the sub-sampling layer** compared to the number of inputs it receives, with each neuron producing a single output from its respective receptive field.
- The degree (amount) of sub-sampling is determined by the size of the receptive fields within these neurons.
 - E.g., Utilizing non-overlapping receptive fields of size 2×2 will reduce the feature map's dimensionality by a factor of two, effectively halving the number of rows and columns in the output compared to the input feature map.

Pooling Function

- Initially, sub-sampling neurons within early convolutional networks would calculate the average of the values within their corresponding area on the feature map, a process known as **average pooling**.
- Contemporary convolutional networks frequently **employ a max pooling** strategy, wherein the **maximum value** from the feature map within a neuron's receptive field is selected.
 - The process of utilizing the max function to condense the information within a receptive field is widely known as **max pooling**.
- The max pooling method is advantageous for its simplicity and effectiveness in **retaining the most prominent features while reducing dimensionality**.

Example: Applying Max Pooling to Feature Map



The result of applying max pooling to the feature map using non-overlapping local receptive fields with a dimensionality of 2-by-2.

Example: CCN Architecture and its Data Flow

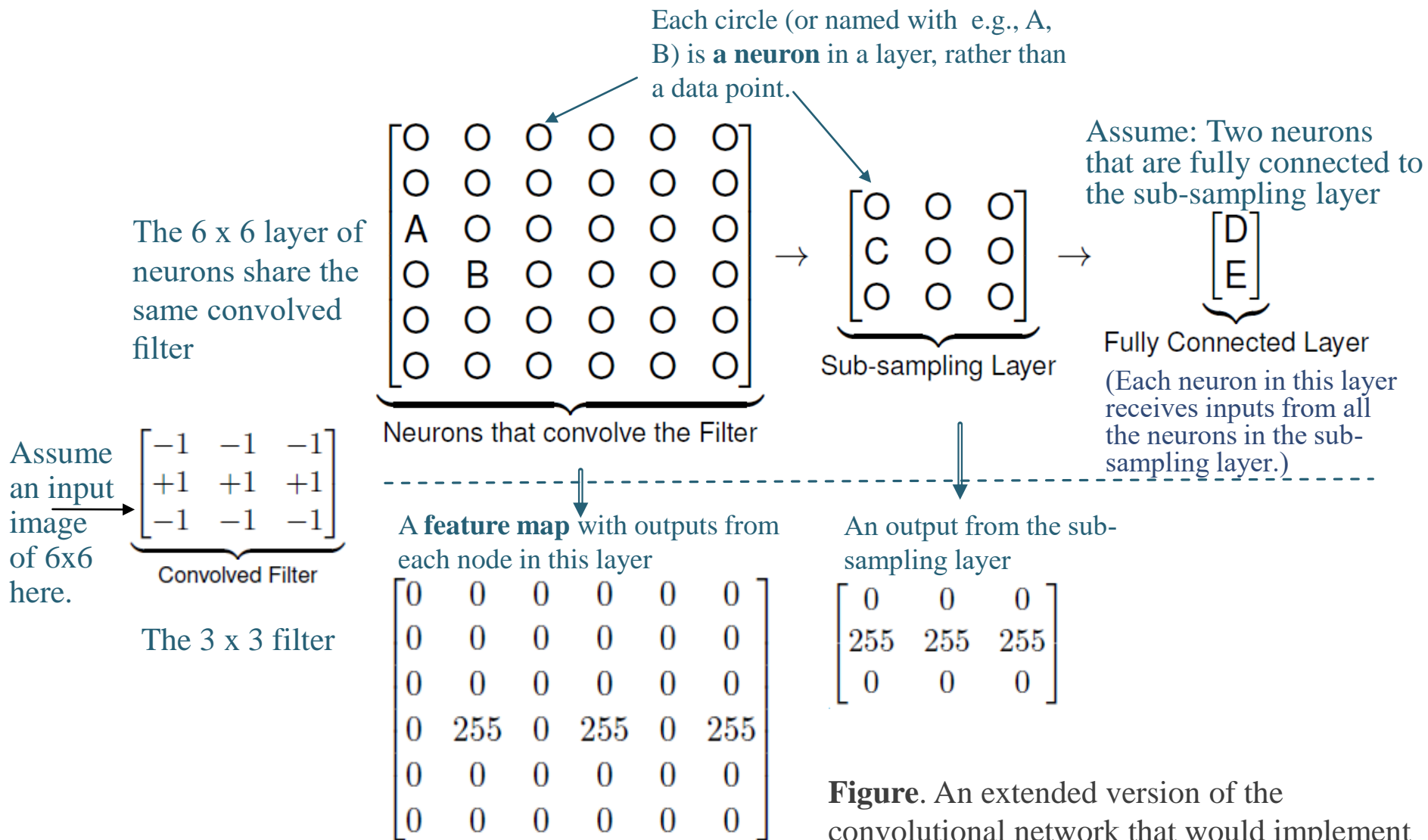


Figure. An extended version of the convolutional network that would implement the data processing with pooling

Outline

- Convolutional Neural Networks
 - Introduction to Convolutional Neural Networks
 - Weight Sharing and Translation Equivalent Feature Detection
 - Filter hyper-parameters: Dimensions, stride and padding
 - Pooling
- 👉 **Training CNNs – Weight Update**

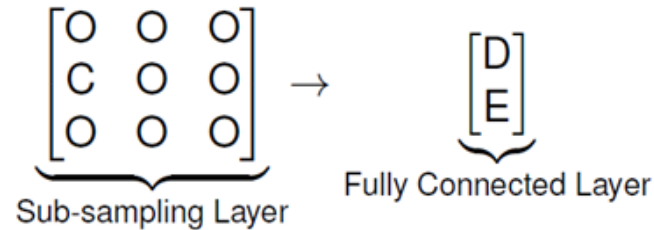
Backward Pass with Max Pooling

- The implementation of a max pooling function in a neural network **requires specific adjustments to the backpropagation process** due to its unique behavior.
- The max function selectively transmits only the highest value among its inputs to the subsequent layer. Hence, **during backpropagation, only this maximum value has a direct influence on the network's learning**, as it is the only one that contributed to the forward pass.
- Consequently, **no gradient is propagated back through the non-maximum inputs** since they did not influence the output (and the error) during the forward pass.
- These values receive a gradient of zero, indicating that they have no impact on the error signal and should not be adjusted during the learning process.

Error Gradient of Neurons in Sub-Sampling Layer

- The **error gradient δ for a neuron in the sub-sampling** is calculated using the following procedure:
 1. Identify the maximum value propagated forward during the pooling process that involved the neuron.
 2. Assign the full error gradient from the max value to the **neuron**, because the gradient of the activation function $\frac{\partial a_i}{\partial z_i}$ of the neuron is 1 for the neuron that contributed the max value.
 - Since max pooling is applied, only the neuron that had the maximum activation in the forward pass (during pooling) will have a non-zero gradient.
 3. For all other neurons that did not contribute to the max value, set the error gradient to zero, as they did not affect the outcome in the forward pass.

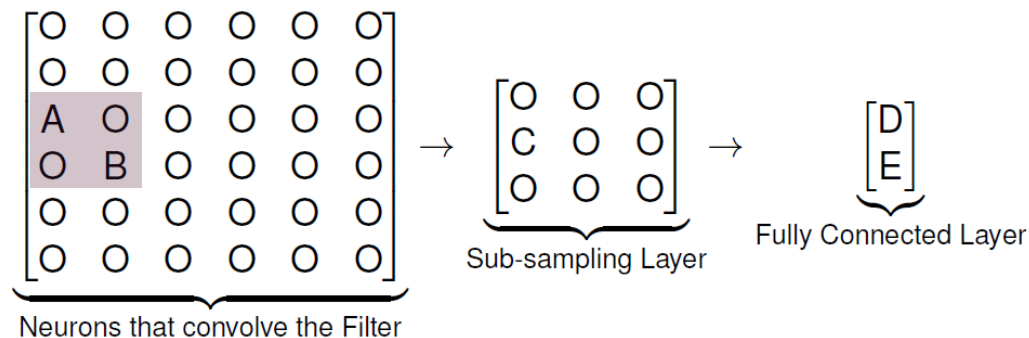
Example: Error Gradient of Neurons in Sub-sampling layer



- **Example:** The δ for Neuron C in a sub-sampling layer with max pooling is:
$$\delta_C = \frac{\partial \varepsilon}{\partial a_C} \times \frac{\partial a_C}{\partial z_C}$$
$$= ((\delta_D \times w_{D,C}) + (\delta_E \times w_{E,C})) \times 1$$
- Assume δ_D and δ_E have already been calculated.
- The gradient of the activation $\frac{\partial a_c}{\partial z_c} = 1$.

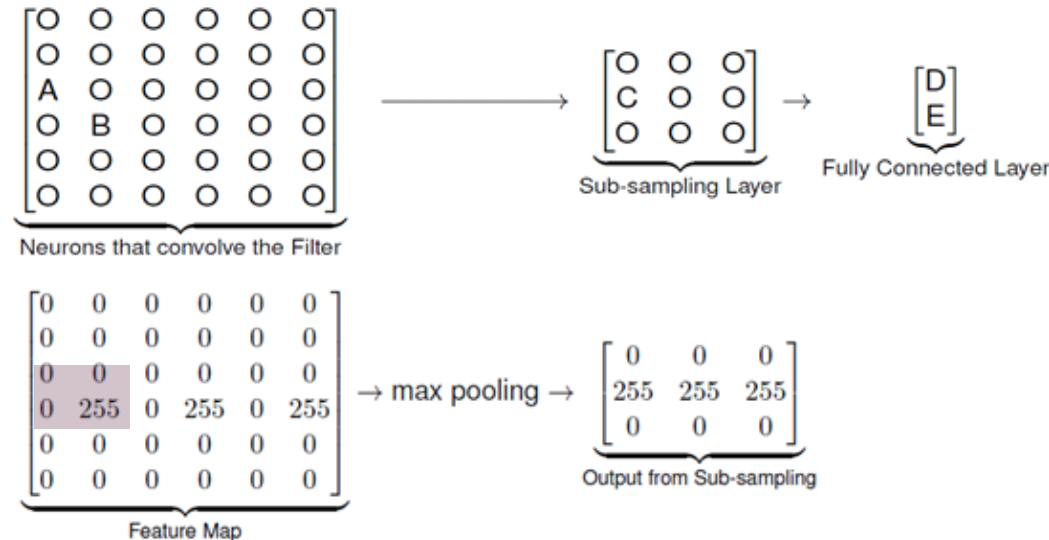
Error Gradient of Neurons in Convolutional Layer

- After determining the error gradient δ for each neuron in the sub-sampling layer, we **backpropagate these values to the preceding convolutional layer**.
- Each neuron in the convolutional layer is uniquely linked to a distinct neuron in the sub-sampling layer.
 - Recall that in a sub-sampling layer, typically implemented as max-pooling, neurons have non-overlapping receptive fields.
 - **Example:** Consider neurons A and B within the receptive field of Neuron C. Neither A nor B feeds forward into any of the other neurons in the sub-sampling layer.



Example: Error Gradient of Neurons in Convolutional Layer

- Calculation of error gradient δ_A of neuron A with non max activation

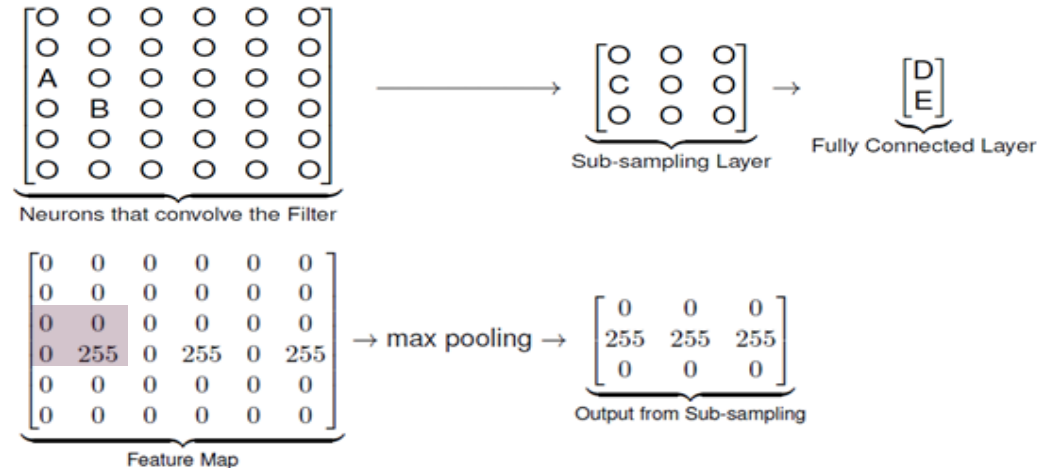


- Given the activations in the feature map, where $a_A = 0$ and $a_B = 255$, Neuron B has the maximum activation within the local receptive field of Neuron C.
- During backpropagation, its error gradient $\delta_A = 0$, since Neuron A did not contribute to the maximum activation,
 - Neuron A did not influence the output during the forward pass in the max-pooling process.

Example: Error Gradient of Neurons in Convolutional Layer

- Calculation of error gradient δ_B of neuron B with the max activation

$$\begin{aligned}\delta_B &= \frac{\partial \varepsilon}{\partial a_B} \times \frac{\partial a_B}{\partial z_B} \\ &= (\delta_D \times w_{C,B}) \times \frac{\partial a_B}{\partial z_B} \\ &= (\delta_D \times 1) \times 1\end{aligned}$$



- $w_{C,B} = 1$ because the max pooling operation does not apply distinct weights to its inputs; effectively, all inputs are considered with equal weight in determining the maximum.
- Neuron B applies a ReLU activation function, therefore $\frac{\partial a_B}{\partial z_B} = 1$, given that $a_B > 0$
- The δ s for other neurons in the layer will be 0 if they did not have the maximum activation within the receptive field of the corresponding neuron in the sub-sampling layer, similar to Neuron A; otherwise, their deltas will be calculated similarly to Neuron B.

Shared Weight Update

- During the training algorithm's backward pass, once the δ for each of the neurons in the first layer has been calculated, we compute an individual weight update for each neuron employing the weight, and then aggregate these individual updates for the ultimate weight update .
- By summing the weight updates across the neurons utilizing it, we maintain a uniform weight across all neurons.
 - This process is similar to the method of aggregating the weight updates for a single weight during batch training;
 - The key distinction being that here is that for each training instance, we sum over the weight updates for every neuron utilizing the weight, rather than combining weight updates across various training instances.

Shared Weight Update

- The combined weight update for neurons sharing a weight.

$$\Delta w_{i,*} = \sum_{i=1}^m \delta_i \times a_*$$

$$w_{i,*} \leftarrow w_{i,*} - \alpha \times \Delta w_{i,*}$$

- Here, m presents the total number of neurons that share the same weight
 - The wildcard $*$ acts as a placeholder for the index corresponding the appropriate input neuron for each neuron utilizing that weight.
-
- During batch gradient descent, this summing process encompasses an additional summation over all the training instances within the batch.