Deap Learning Training

CSI 5180 - Machine Learning for Bioinformatics

Marcel Turcotte

Version: Mar 14, 2025 11:42

Preamble

Quote of the Day

nature machine intelligence

Review article https://doi.org/10.1038/s42256-025-01007-9

Transformers and genome language models

Received: 29 January 2024

Accepted: 31 January 2025

Published online: 13 March 2025

Check for updates

Micaela E. Consens[1,2,3], Cameron Dufault[1], Michael Wainberg[2,4,5,6,7], Duncan Forster[2,8,9], Mehran Karimzadeh[2,10,11,12], Hani Goodarzi[10,11,12], Fabian J. Theis[13,14,15,16], Alan Moses[1,17] & Bo Wang[1,2,3,18]

Large language models based on the transformer deep learning architecture have revolutionized natural language processing. Motivated by the analogy between human language and the genome's biological code, researchers have begun to develop genome language models (gLMs) based on transformers and related architectures. This Review explores the use of transformers and language models in genomics. We survey open questions in genomics amenable to the use of gLMs, and motivate the use of gLMs and the transformer architecture for these problems. We discuss the potential of gLMs for modelling the genome using unsupervised pretraining tasks, specifically focusing on the power of zero- and few-shot learning. We explore the strengths and limitations of the transformer architecture, as well as the strengths and limitations of current gLMs more broadly. Additionally, we contemplate the future of genomic modelling beyond the transformer architecture, based on current trends in research. This Review serves as a guide for computational biologists and computer scientists interested in transformers and language models for genomic data.

Consens et al. (2025)

Summary

In this lecture, we will explore the fundamentals of **Convolutional Neural**

Networks (CNNs) and their role in deep learning. We will begin by discussing the **hierarchical structure** of deep learning models and the advantages of deep networks over shallow ones. The lecture will cover convolutional operations, including **kernels**, **receptive fields**, **padding**, and **stride**, and explain their significance in feature extraction. Additionally, we will examine the importance of **pooling layers** in reducing dimensionality and improving computational efficiency. By the end of the session, students will gain a solid understanding of CNN architectures and their applications in **bioinformatics** and beyond.

Learning Objectives

Upon completing this lecture, students will be able to:

- **Explain** the hierarchical representation of concepts in deep learning.
- **Compare** deep and shallow neural networks in terms of efficiency and expressiveness.
- **Describe** the structure and function of Convolutional Neural Networks (CNNs).
- **Understand** convolution operations, including kernels and feature extraction.
- **Explain** the concepts of receptive fields, padding, and stride in CNNs.
- **Discuss** the role and benefits of pooling layers in CNN architectures.
- **Apply** CNNs in bioinformatics and recognize their impact on genomic analysis.

Today, we have a particularly dense agenda. The study of convolutional networks involves multiple levels of complexity. Please feel free to ask questions if you need clarification.

Detailed learning objectives.

1. **Explain the Hierarchy of Concepts in Deep Learning**
    - Understand how deep learning models build hierarchical representations of data.
    - Recognize how this hierarchy reduces the need for manual feature engineering.
2. **Compare Deep and Shallow Neural Networks**
    - Discuss why deep networks are more parameter-efficient than shallow networks.
    - Explain the benefits of depth in neural network architectures.
3. **Describe the Structure and Function of Convolutional Neural Networks (CNNs)**

- Understand how CNNs detect local patterns in data.
- Explain how convolutional layers reduce the number of parameters through weight sharing.

4. **Understand Convolution Operations Using Kernels**
   - Describe how kernels (filters) are applied over input data to perform convolutions.
   - Explain how feature maps are generated from convolution operations.
5. **Explain Receptive Fields, Padding, and Stride in CNNs**
   - Define the concept of a receptive field in convolutional layers.
   - Understand how padding and stride affect the output dimensions and computation.
6. **Discuss the Role and Benefits of Pooling Layers**
   - Explain how pooling layers reduce spatial dimensions and control overfitting.
   - Describe how pooling introduces translation invariance in CNNs.

Foreword

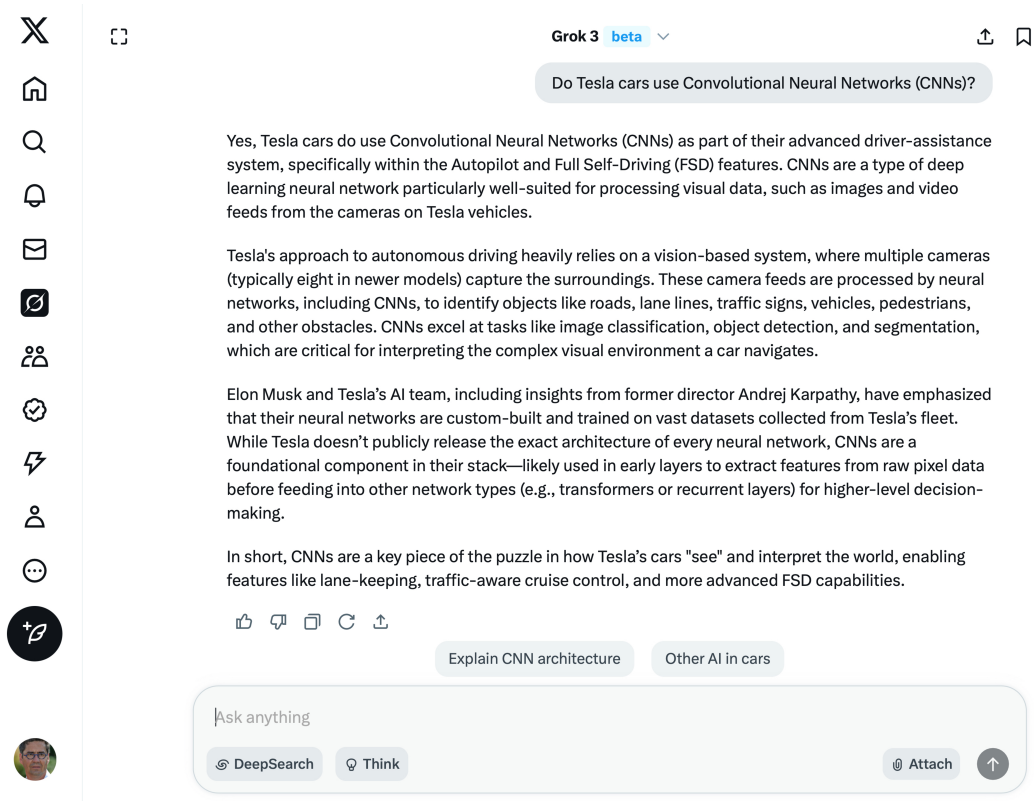
- Convolutional Neural Networks (CNNs) have **significantly contributed** to the advancement and integration of deep learning technologies.
- They are extensively applied in numerous **life science domains**, as evidenced in research such as Zeng et al. (2016) and He et al. (2020).
- CNN architectures are categorized into **1D**, **2D**, and **3D** models.
- Although 1D convolutions offer simplicity, this discussion will commence with **2D convolutions** due to their **prevalent** use and to illustrate the conceptual **hierarchy** inherent in these models.

Introduction

Convolutional Neural Networks

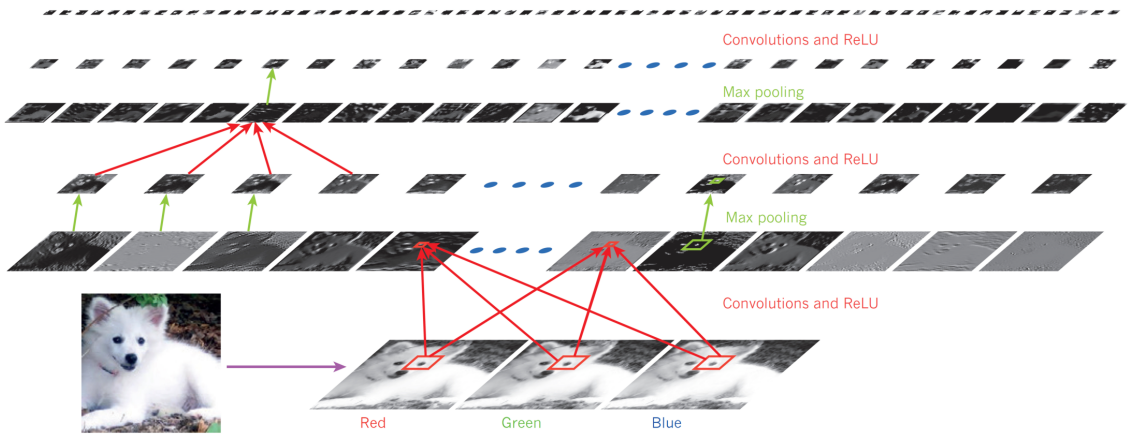
Yann LeCun, recognized as one of the three pioneers of deep learning and the inventor of Convolutional Neural Networks (CNNs), frequently engages in discussions with Elon Musk on the social media platform X (previously known as Twitter).

Who Holds the Truth?



Grok Interaction on March 13, 2025

Hierarchy of Concepts



Attribution: LeCun, Bengio, and Hinton (2015)

In the book “Deep Learning” (Goodfellow, Bengio, and Courville 2016), authors Goodfellow, Bengio, and Courville define deep learning as a subset of machine learning that enables computers to “understand the world in terms of a hierarchy of concepts.”

This hierarchical approach is one of deep learning’s most significant contributions. It reduces the need for manual feature engineering and redirects

the focus toward the engineering of neural network architectures.

Convolutional Neural Networks (CNNs) have had a profound impact on the field of machine learning, particularly in areas involving image and video processing.

1. **Revolutionizing Image Recognition**: CNNs have significantly advanced the state of the art in image recognition and classification, achieving high accuracy across various datasets. This has led to breakthroughs in fields such as medical imaging, autonomous vehicles, and facial recognition.
2. **Feature Extraction**: CNNs automatically learn to extract features from raw data, eliminating the need for manual feature engineering. This capability has been crucial in handling complex data patterns and has expanded the applicability of machine learning to diverse domains.
3. **Transfer Learning**: CNNs facilitate transfer learning, where pre-trained networks on large datasets can be fine-tuned for specific tasks with limited data. This has made CNNs accessible and effective for a wide range of applications beyond their original training scope.
4. **Advancements in Deep Learning**: The success of CNNs has spurred further research in deep learning architectures, inspiring the development of more sophisticated models like recurrent neural networks (RNNs), long short-term memory networks (LSTMs), and transformer models.
5. **Broader Application Areas**: Beyond image processing, CNNs have been adapted for natural language processing, audio processing, and even in bioinformatics for tasks such as protein structure prediction and genomics.
6. **Implications for Real-World Applications**: CNNs have enabled practical applications in fields such as healthcare, where they assist in diagnostic imaging, and in security, where they enhance surveillance systems. They have also contributed to advancements in virtual reality, gaming, and augmented reality.

Hierarchy of Concepts

- Each layer detects **patterns** from the output of the **layer preceding it**.
  - In other words, proceeding from the input to the output of the network, the network uncovers "**patterns of patterns**".
    - Analyzing an image, the networks first detect simple patterns, such as **vertical**, **horizontal**, **diagonal** lines, **arcs**, etc.

        - These are then combined to form **corners**, **crosses**, etc.
- (This illustrates how **transfer learning** works and why retaining the bottom layers only.)

But Also …

> "An MLP with just **one hidden layer** can theoretically model even the most **complex functions**, provided it **has enough neurons**. But for complex problems, **deep networks** have a much **higher parameter efficiency** than shallow ones: they can model complex functions **using exponentially fewer neurons** than shallow nets, allowing them to reach much **better performance** with the same amount of training data."

Géron (2019) § 10

During the lecture, attempt to discern why convolutional neural networks possess fewer parameters compared to fully connected feedforward networks.

How Many Layers?

- Start with one layer, then **increase the number of layers** until the model starts **overfitting** the training data.
- **Finetune** the model adding regularization (dropout layers, regularization terms, etc.).

The number of neurons and other hyperparameters are determined using a grid search.

Observations

Consider a **feed-forward network** (FFN) and its model:

$$ h_{W,b}(X) = \phi_k(\ldots \phi_2(\phi_1(X)) \ldots) $$

where

$$ \phi_l(Z) = \sigma(W_l Z + b_l) $$

for $l=1 \ldots k$. - The **number of parameters** in grows rapidly:

$$ (\text{size of layer}_{l-1} + 1) \times \text{size of layer}_{l} $$

Image Classification Task

- Consider an **RGB image** with dimensions 224×224, which is relatively small by contemporary benchmarks.
- The image consists of $224 \times 224 \times 3 = 150,528$ **input features**.
- A neural network with merely a **single hidden dense layer** would require over 22,658,678,784 (**22 billion**) parameters, highlighting the computational complexity involved.

Observations (Continued)

- Crucial pattern information is often **local**.
  - e.g., edges, corners, crosses.
- **Convolutional layers** reduce parameters significantly.
  - Unlike dense layers, neurons in a convolutional layer are **not fully connected** to the preceding layer.
  - Neurons connect only within their **receptive fields** (rectangular regions).

Convolutional networks originate from the domain of **machine vision**, which explains their intrinsic compatibility with **grid-structured inputs**.

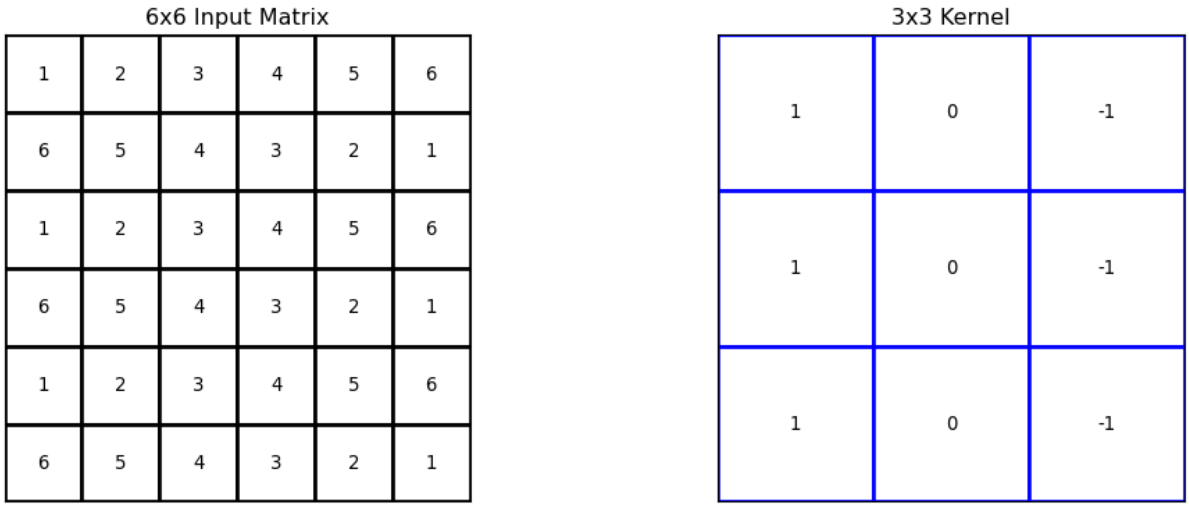
The original publication by Yann Lecun has been cited nearly 35,000 times (Lecun et al. 1998).

Kernel

Kernel

In [2]:
```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.patches as patches

def plot_matrix(ax, matrix, title="Matrix", edge_color='black', face_c
    ax.matshow(matrix, cmap='gray', alpha=0.2)
    for (i, j), val in np.ndenumerate(matrix):
        ax.text(j, i, f'{val}', ha='center', va='center', color='black

        # Highlight specific region if provided
        if highlight_region and (i, j) in highlight_region:
            rect_face_color = 'yellow'
```

```
        else:
            rect_face_color = face_color

        ax.add_patch(patches.Rectangle((j-0.5, i-0.5), 1, 1, fill=True

    ax.set_xticks([])
    ax.set_yticks([])
    ax.set_title(title)

def apply_kernel(matrix, kernel):
    kernel_size = kernel.shape[0]
    output_size = matrix.shape[0] - kernel_size + 1
    output = np.zeros((output_size, output_size), dtype=int)

    for i in range(output_size):
        for j in range(output_size):
            sub_matrix = matrix[i:i+kernel_size, j:j+kernel_size]
            output[i, j] = np.sum(sub_matrix * kernel)
    return output

# Define a 6x6 matrix and a 3x3 kernel
matrix = np.array([
    [1, 2, 3, 4, 5, 6],
    [6, 5, 4, 3, 2, 1],
    [1, 2, 3, 4, 5, 6],
    [6, 5, 4, 3, 2, 1],
    [1, 2, 3, 4, 5, 6],
    [6, 5, 4, 3, 2, 1]
])

kernel = np.array([
    [1, 0, -1],
    [1, 0, -1],
    [1, 0, -1]
])

# Apply the kernel to the matrix
output = apply_kernel(matrix, kernel)

# Plot the original matrix, kernel, and the result
fig, axes = plt.subplots(1, 2, figsize=(12, 4))
plot_matrix(axes[0], matrix, "6x6 Input Matrix")
plot_matrix(axes[1], kernel, "3x3 Kernel", edge_color='blue')
# plot_matrix(axes[2], output, "Output Matrix", edge_color='green')

plt.tight_layout()
plt.show()
```

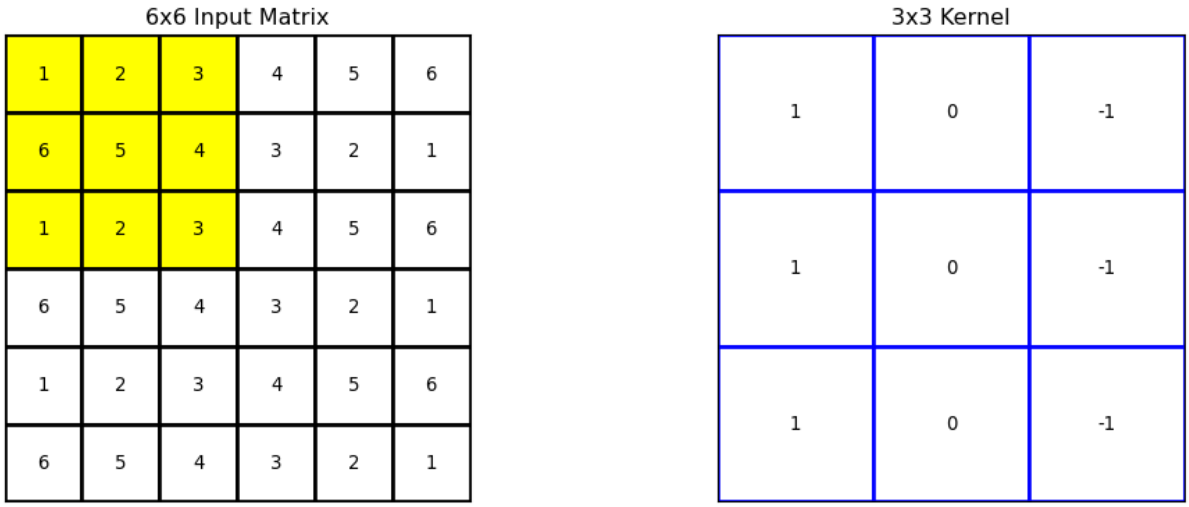


A **kernel** is a small matrix, usually 3×3, 5×5, or similar in size, that slides over the input data (such as an image) to perform convolution.

Firstly, we will examine the technical components of convolution, focusing on the interaction between the input matrix and the kernel. Subsequently, we will explore the computational process involved in performing a convolution. Lastly, we will analyze the significance and implications of convolution.

Kernel

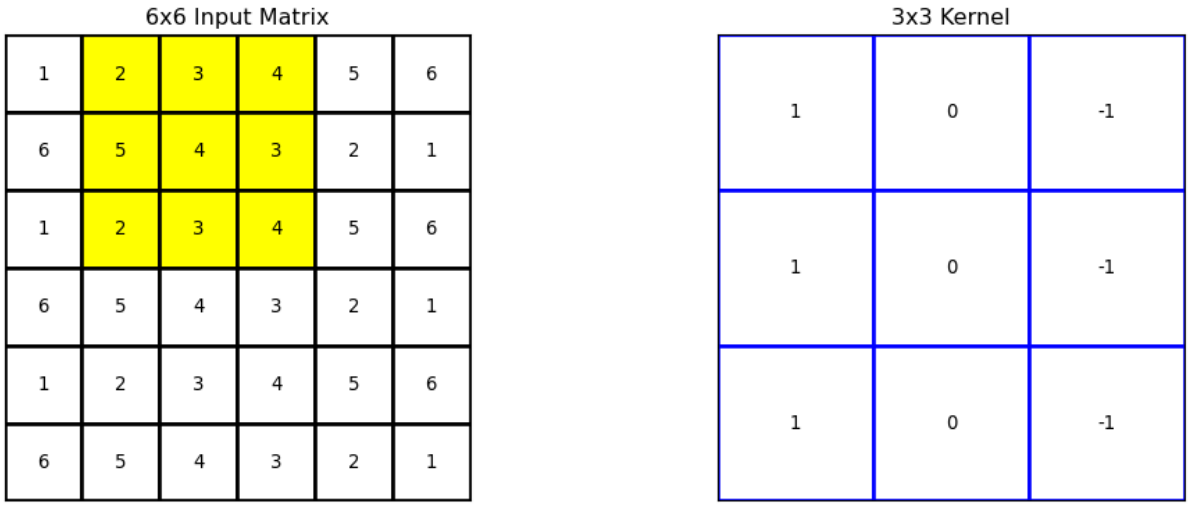
In [3]:
```
# Apply the kernel to the matrix
output = apply_kernel(matrix, kernel)

# Define the region to highlight (top-left 3x3 cells)
highlight_region = [(i, j) for i in range(3) for j in range(3)]

# Plot the original matrix, kernel, and the result
fig, axes = plt.subplots(1, 2, figsize=(12, 4))
plot_matrix(axes[0], matrix, "6x6 Input Matrix", highlight_region=high
plot_matrix(axes[1], kernel, "3x3 Kernel", edge_color='blue')

plt.tight_layout()
plt.show()
```



Beginning with the kernel positioned to overlap the upper-left corner of the input matrix.

Kernel

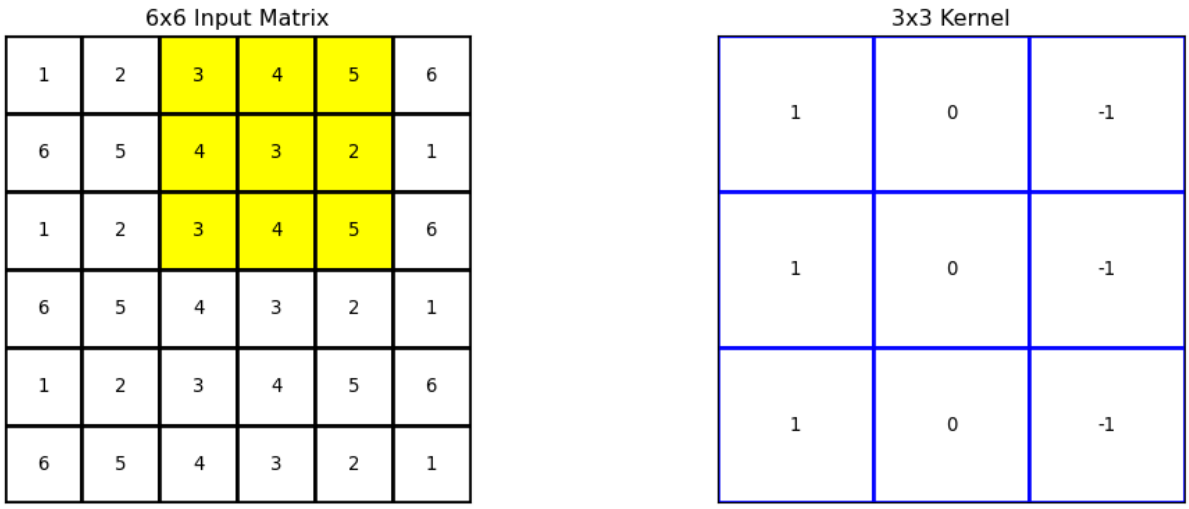
In [4]:
```
# Apply the kernel to the matrix
output = apply_kernel(matrix, kernel)

# Define the region to highlight (top-left 3x3 cells)
highlight_region = [(i, j) for i in range(3) for j in range(1,4)]

# Plot the original matrix, kernel, and the result
fig, axes = plt.subplots(1, 2, figsize=(12, 4))
plot_matrix(axes[0], matrix, "6x6 Input Matrix", highlight_region=high
plot_matrix(axes[1], kernel, "3x3 Kernel", edge_color='blue')

plt.tight_layout()
plt.show()
```



It can be moved to the right three times.

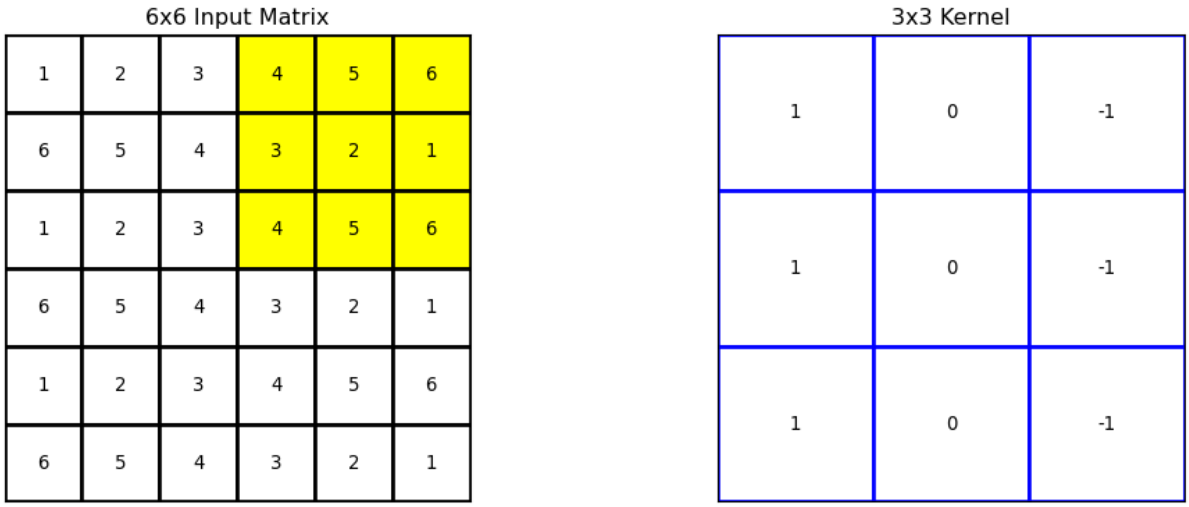
Kernel

In [5]:
```
# Apply the kernel to the matrix
output = apply_kernel(matrix, kernel)

# Define the region to highlight (top-left 3x3 cells)
highlight_region = [(i, j) for i in range(3) for j in range(2,5)]

# Plot the original matrix, kernel, and the result
fig, axes = plt.subplots(1, 2, figsize=(12, 4))
plot_matrix(axes[0], matrix, "6x6 Input Matrix", highlight_region=high
plot_matrix(axes[1], kernel, "3x3 Kernel", edge_color='blue')

plt.tight_layout()
plt.show()
```



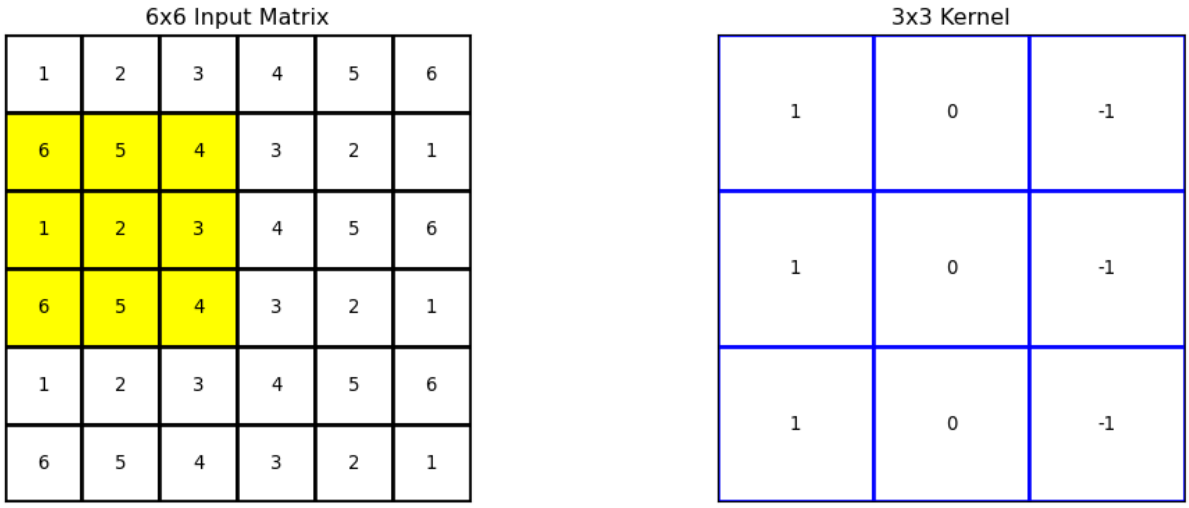
Kernel

In [6]:
```
# Apply the kernel to the matrix
output = apply_kernel(matrix, kernel)

# Define the region to highlight (top-left 3x3 cells)
highlight_region = [(i, j) for i in range(3) for j in range(3,6)]

# Plot the original matrix, kernel, and the result
fig, axes = plt.subplots(1, 2, figsize=(12, 4))
plot_matrix(axes[0], matrix, "6x6 Input Matrix", highlight_region=high
plot_matrix(axes[1], kernel, "3x3 Kernel", edge_color='blue')

plt.tight_layout()
plt.show()
```



Kernel

In [7]:
```
# Apply the kernel to the matrix
output = apply_kernel(matrix, kernel)

# Define the region to highlight (top-left 3x3 cells)
highlight_region = [(i, j) for i in range(1,4) for j in range(3)]

# Plot the original matrix, kernel, and the result
fig, axes = plt.subplots(1, 2, figsize=(12, 4))
plot_matrix(axes[0], matrix, "6x6 Input Matrix", highlight_region=high
plot_matrix(axes[1], kernel, "3x3 Kernel", edge_color='blue')

plt.tight_layout()
plt.show()
```



The kernel can then be moved to the second row of the input matrix, and moved to the right three times.

How many placements of the kernel over the input matrix are there? $4 \times 4 =

16$.

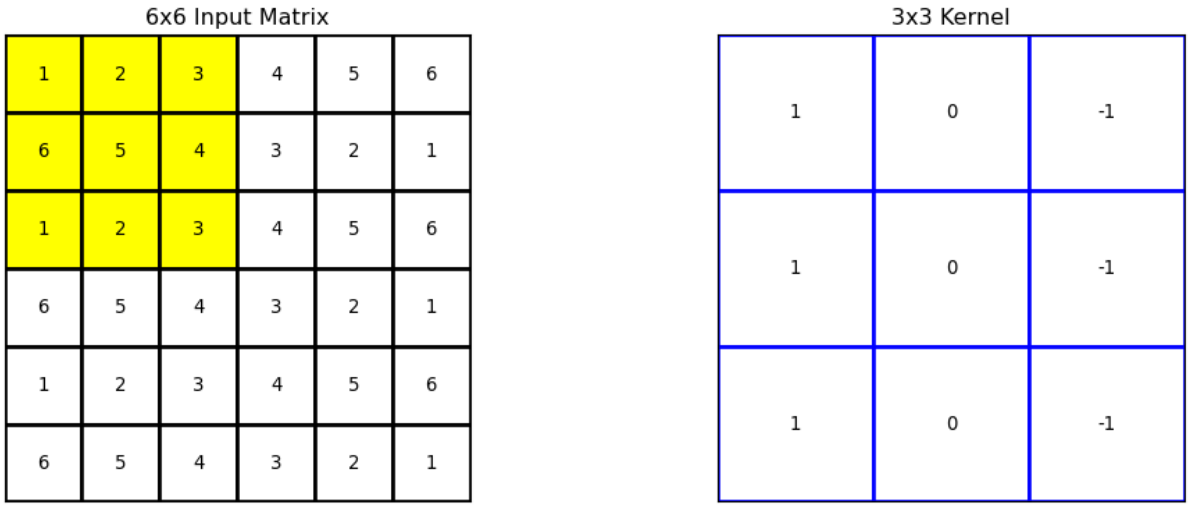
Kernel Placements

In [8]:
```
# Apply the kernel to the matrix
output = apply_kernel(matrix, kernel)

fig, axes = plt.subplots(4, 4, figsize=(12, 12))

for row in range(4):

  for col in range(4):

    # Define the region to highlight (top-left 3x3 cells)
    highlight_region = [(i, j) for i in range(row, row+3) for j in ran

    # Plot the original matrix, kernel, and the result

    plot_matrix(axes[row,col], matrix, "6x6 Input Matrix", highlight_r

plt.tight_layout()
plt.show()
```


6x6 Input Matrix
1 2 3 4 5 6
6 5 4 3 2 1
1 2 3 4 5 6
6 5 4 3 2 1
1 2 3 4 5 6
6 5 4 3 2 1

Kernel

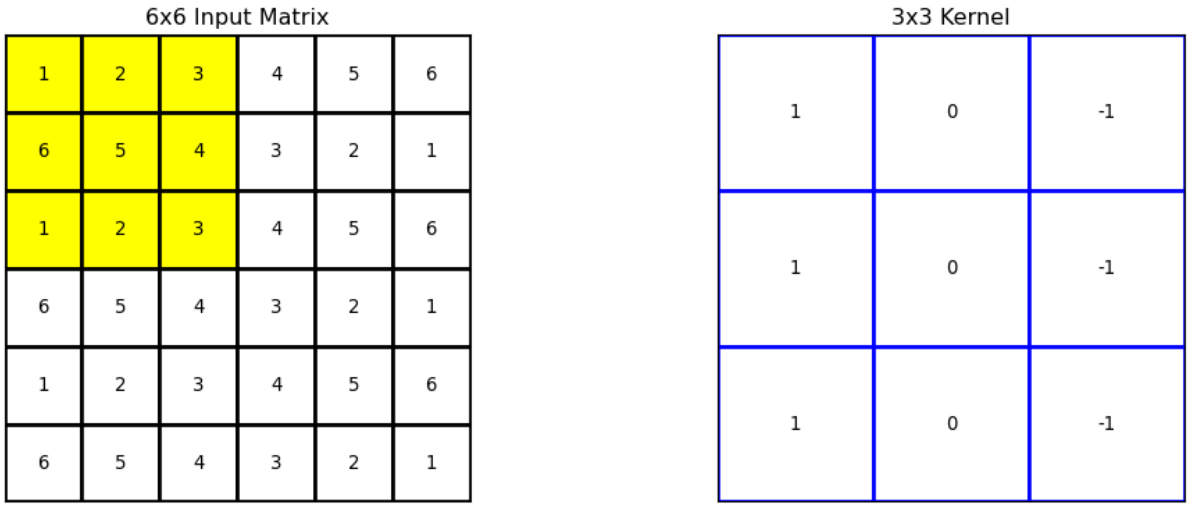
In [9]:



With the kernel placed over a specific region of the input matrix, the **convolution** is **element-wise multiplication** (each element of the kernel is multiplied by the corresponding element of the input matrix region it overlaps) followed by a **summation** of the results to produce a **single scalar value**.

Kernel

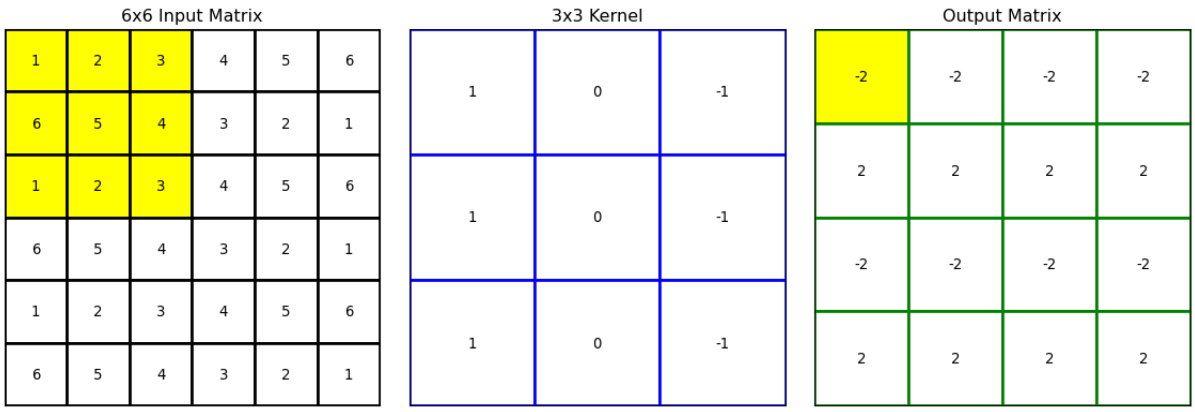
In [10]:
```
# Apply the kernel to the matrix
output = apply_kernel(matrix, kernel)

# Define the region to highlight (top-left 3x3 cells)
highlight_region = [(i, j) for i in range(3) for j in range(3)]

# Plot the original matrix, kernel, and the result
fig, axes = plt.subplots(1, 2, figsize=(12, 4))
plot_matrix(axes[0], matrix, "6x6 Input Matrix", highlight_region=high
plot_matrix(axes[1], kernel, "3x3 Kernel", edge_color='blue')

plt.tight_layout()
plt.show()
```



$1 \times 1 + 2 \times 0 + 3 \times (-1) + 6 \times 1 + 5 \times 0 + 4 \times (-1) + 1 \times 1 + 2 \times 0 + 3 \times (-1) = -2$

Kernel

In [11]:



The 16 resulting values can be organized into an **output matrix**. The element at position (0,0) in this output matrix represents the result of applying the convolution operation with the kernel at the initial position on the input matrix. In convolutional neural networks, the output matrix is referred to as a **feature map**.

It is referred to as a feature map because these outputs serve as features for the subsequent layer. In CNNs, the term “feature map” refers to the output of a convolutional layer after applying filters to the input data. These feature maps capture various patterns or features from the input, such as edges or textures in image data.

The output feature maps of one layer become the input for the next layer, effectively serving as features that the subsequent layer can use to learn more

complex patterns. This hierarchical feature extraction process is a key characteristic of CNNs, allowing them to build progressively more abstract and high-level representations of the input data as the network depth increases.

Blurring

In [12]:
```
import numpy as np
from PIL import Image
import matplotlib.pyplot as plt
from scipy.ndimage import convolve

def apply_kernel_to_image(image_path, kernel):

    # Load the image and convert it to grayscale
    image = Image.open(image_path).convert('L')
    image_array = np.array(image)

    # Apply the convolution using the provided kernel
    convolved_array = convolve(image_array, kernel, mode='reflect')

    # Convert the convolved array back to an image
    convolved_image = Image.fromarray(convolved_array)

    # Display the original and convolved images
    plt.figure(figsize=(10, 4))

    plt.subplot(1, 2, 1)
    plt.title('Original Image')
    plt.imshow(image_array, cmap='gray')
    plt.axis('off')

    plt.subplot(1, 2, 2)
    plt.title('Convolved Image')
    plt.imshow(convolved_image, cmap='gray')
    plt.axis('off')

    plt.tight_layout()
    plt.show()

# Define the 7x7 averaging kernel

kernel = np.array([
    [1/49, 1/49, 1/49, 1/49, 1/49, 1/49, 1/49],
    [1/49, 1/49, 1/49, 1/49, 1/49, 1/49, 1/49],
    [1/49, 1/49, 1/49, 1/49, 1/49, 1/49, 1/49],
    [1/49, 1/49, 1/49, 1/49, 1/49, 1/49, 1/49],
    [1/49, 1/49, 1/49, 1/49, 1/49, 1/49, 1/49],
    [1/49, 1/49, 1/49, 1/49, 1/49, 1/49, 1/49],
    [1/49, 1/49, 1/49, 1/49, 1/49, 1/49, 1/49]
])
```

```
# Apply the kernel to the image (provide your image path here)

image_path = '../../assets/images/uottawa_hor_black.png'
apply_kernel_to_image(image_path, kernel)
```

A pixel is transformed into the **average of itself** and **its eight surrounding neighbors**, resulting in a blurred effect on the image.

The application of kernels to images has been a longstanding practice in the field of image processing.

Vertical Edge detection

In [13]:
```
# Define the 3x3 vertical edge detection kernel

kernel = np.array([
    [-1, 0, 1],
    [-2, 0, 2],
    [-1, 0, 1]
])

# Apply the kernel to the image (provide your image path here)

image_path = '../../assets/images/uottawa_hor_black.png'
apply_kernel_to_image(image_path, kernel)
```

This kernel detects vertical edges by **emphasizing differences in intensity between adjacent columns**. It subtracts pixel values on the left from those on the right, enhancing vertical transitions and suppressing uniform regions.

This is a type of **edge detection kernel**, specifically a horizontal gradient filter or a Sobel operator. It's designed to detect changes in intensity along the horizontal axis, emphasizing vertical edges in an image.

When this kernel is convolved with an image:

- It highlights vertical edges by calculating the difference in pixel intensity between the left and right sides of a point.
- The negative values (-1) on the left subtract intensity, while the positive values (1) on the right add intensity, effectively measuring the horizontal gradient.
- The zeros in the middle ignore the central pixel's contribution, focusing only on the contrast between left and right neighbors.

The result is an image where:

- Vertical edges (e.g., the boundary between a dark object and a light background) appear bright or dark, depending on the direction of the intensity change.
- Horizontal edges or uniform areas tend to be suppressed (close to zero).

Imagine sliding this kernel over an image like a scanner. For each pixel:

- It looks at the pixels to its left (subtracting their value with -1) and to its right (adding their value with 1).
- If the left and right sides are similar (e.g., same color), the result is near zero (no edge).
- If the left is dark and the right is light (or vice versa), the result is a strong positive or negative value, showing a vertical edge.

This is useful in computer vision (like in Tesla's CNNs) to help detect object boundaries or lane lines by emphasizing where pixel values change sharply in the horizontal direction.

Horizontal Edge detection

In [14]:
```
# Define the 3x3 horizontal edge detection kernel

kernel = np.array([
    [1, 2, 1],   # Top row (positive)
    [0, 0, 0],   # Middle row (neutral)
    [-1, -2, -1] # Bottom row (negative)
])
# Apply the kernel to the image (provide your image path here)

image_path = '../../assets/images/uottawa_hor_black.png'
apply_kernel_to_image(image_path, kernel)
```

This kernel detects horizontal edges by **highlighting differences in intensity between adjacent rows**. It subtracts pixel values in the upper row from those in the lower row, accentuating horizontal transitions while minimizing uniform areas.

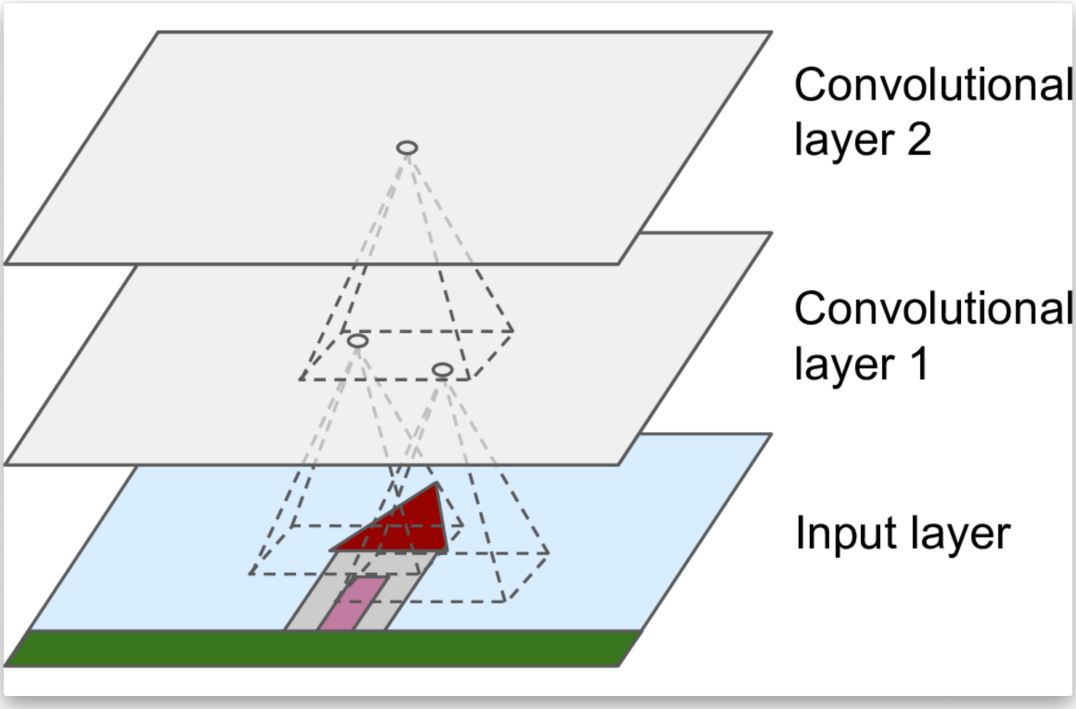
Kernels

In contrast to *image processing*, where *kernels are manually defined by the user*, in **convolutional networks**, the **kernels are automatically learned by the network**.

To be continued $\ldots$

Terminology

Receptive Field



Attribution: Géron (2019) Figure 14.2

Receptive Field

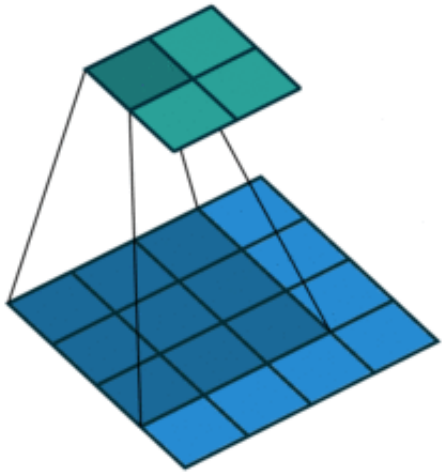
- Each unit is connected to neurons in its **receptive fields**.
  - Unit i,j in layer l is connected to the units i to $i+f_h-1$, j to $j+f_w-1$ of the layer $l-1$, where f_h and f_w are respectively the **height** and **width** of the **receptive field**.

Padding

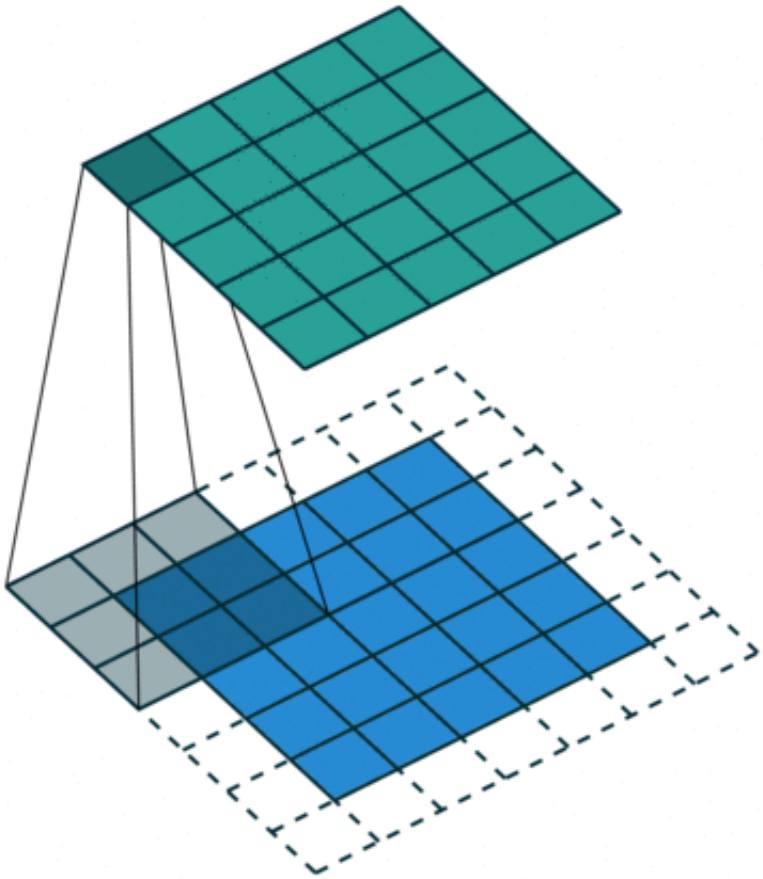
Zero padding. In order to have layers of the same size, the grid can be padded with zeros.

Padding

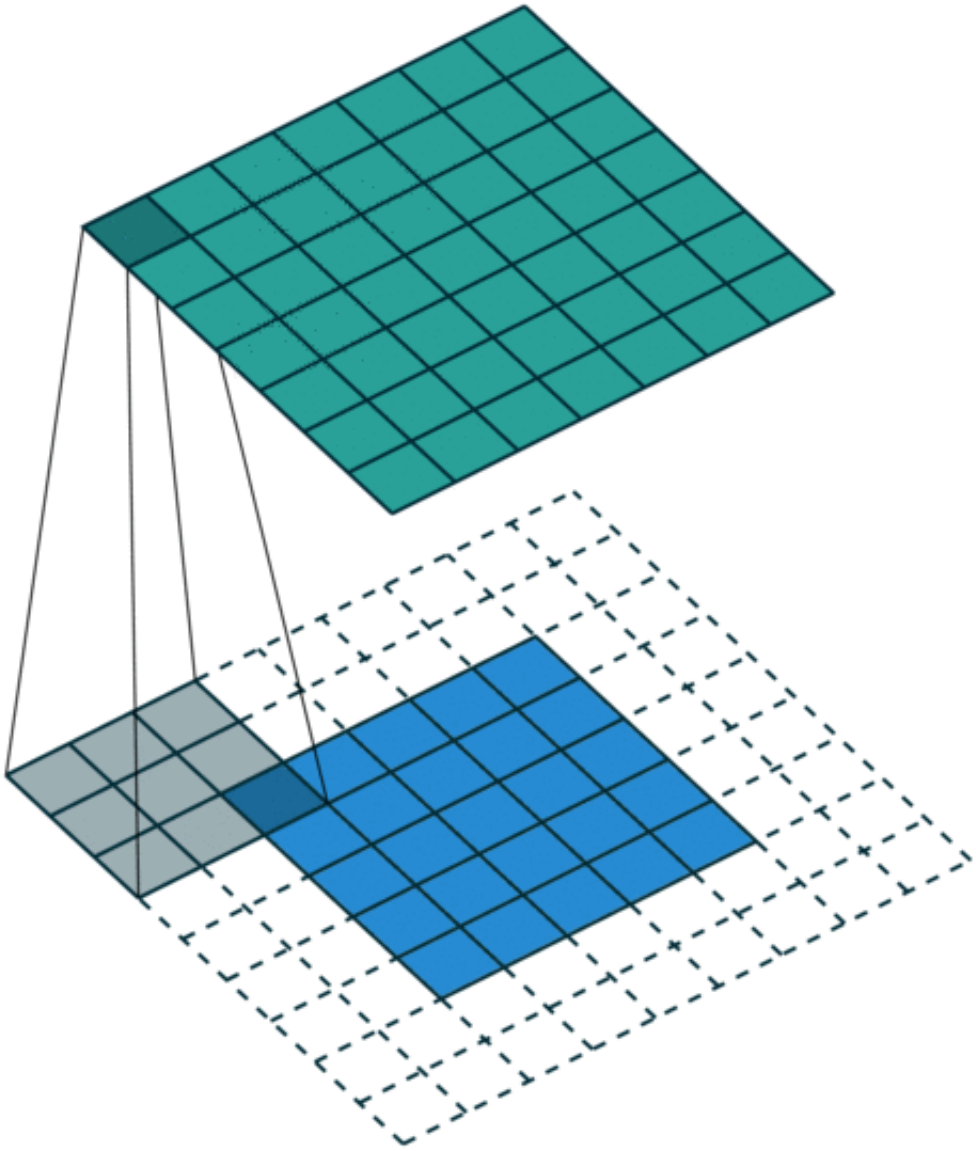
No padding



Half padding



Full padding



Attribution: github.com/vdumoulin/conv_arithmetic

Stride

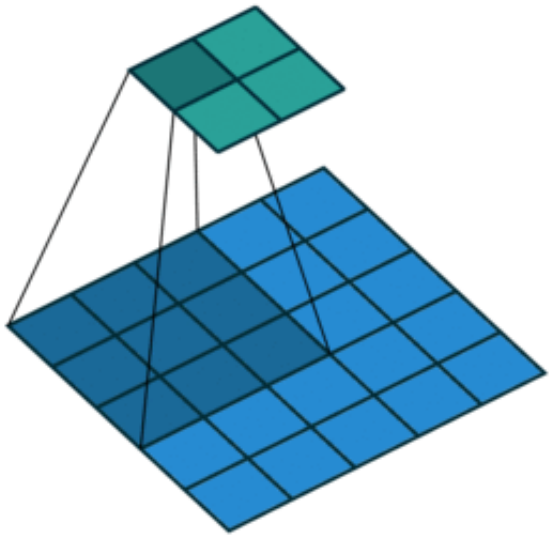
Stride. It is possible to connect a larger layer $(l-1)$ to a smaller one (l) by skipping units. The number of units skipped is called **stride**, s_h and s_w.

. . .

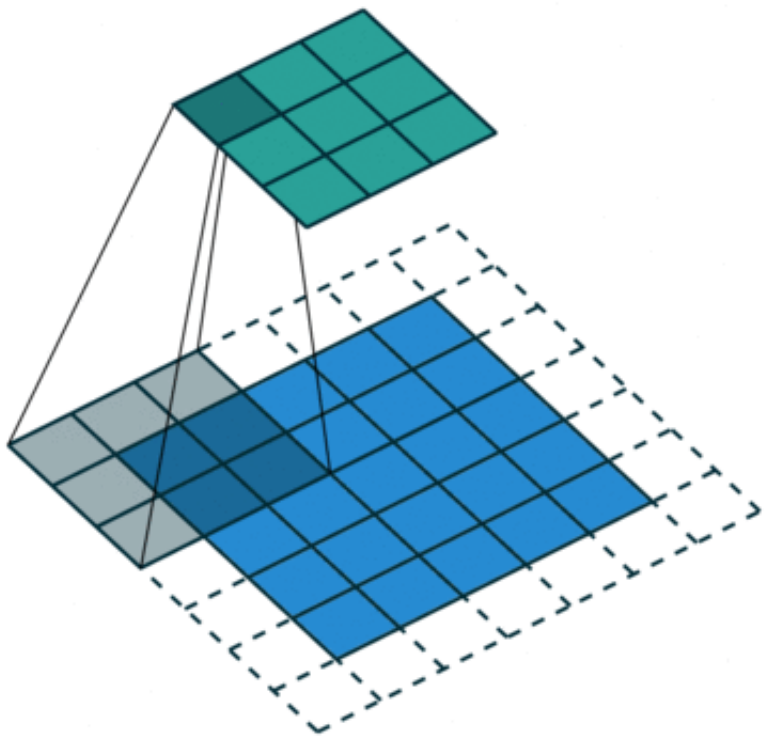
- Unit i,j in layer l is connected to the units $i \times s_h$ to $i \times s_h + f_h - 1$, $j \times s_w$ to $j \times s_w + f_w - 1$ of the layer $l-1$, where f_h and f_w are respectively the **height** and **width** of the **receptive field**, s_h and s_w are respectively the **height** and **width strides**.

Stride

No padding, strides



Padding, strides



Attribution: github.com/vdumoulin/conv_arithmetic

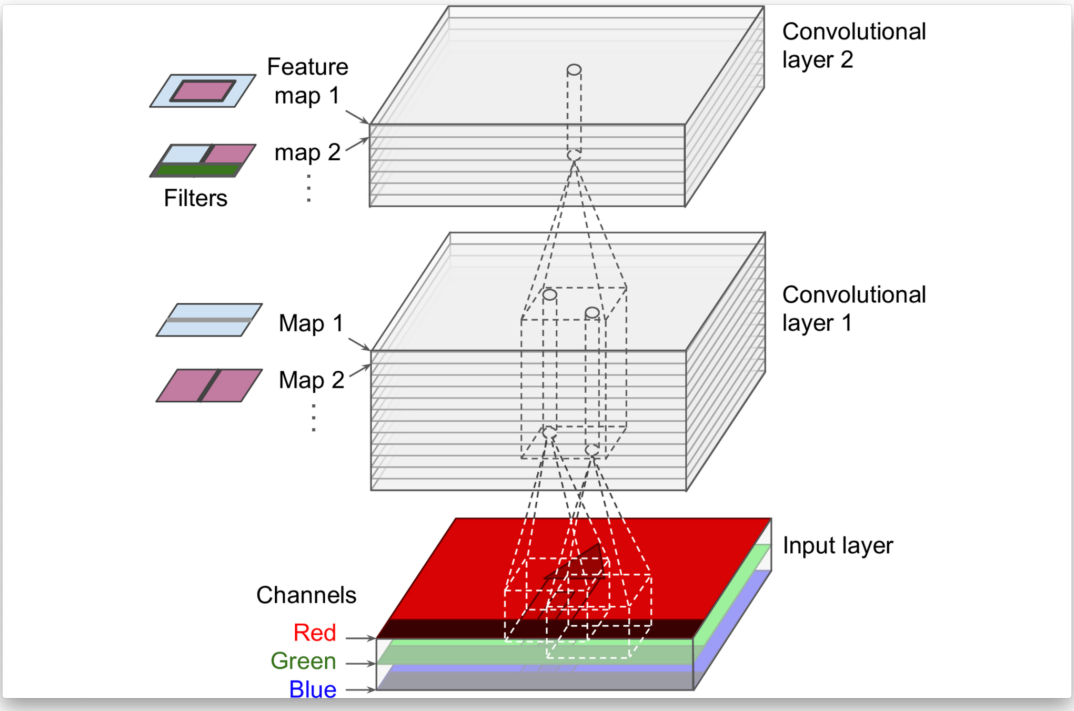
Filters

Filters

- A **window** of size $f_h \times f_w$ is moved over the output of layers $l-1$, referred to as the **input feature map**, position by position.
- **For each location**, the product is calculated between the extracted patch and a matrix of the same size, known as a **convolution kernel** or **filter**. The **sum** of the values in the resulting matrix constitutes the **output** for that location.

Model

Model



Attribution: Géron (2019) Figure 14.6

Convolutional Layer

- "Thus, a layer full of neurons using the **same filter** outputs a **feature map**."

- “Of course, you do not have to define the filters manually: instead, **during training the convolutional layer will automatically learn the most useful filters for its task**.”

Géron (2019) § 14

Convolutional Layer

- “(…) and **the layers above will learn to combine them into more complex patterns**.”
- “The fact that **all neurons in a feature map share the same parameters** dramatically reduces the number of parameters in the model.”

Géron (2019) § 14

Summmary

1. **Feature Map:** In convolutional neural networks (CNNs), the output of a convolution operation is known as a feature map. It captures the features of the input data as processed by a specific kernel.

Summmary

1. **Kernel Parameters:** The parameters of the kernel are learned through the backpropagation process, allowing the network to optimize its feature extraction capabilities based on the training data.

Summmary

1. **Bias Term:** A single bias term is added uniformly to all entries of the feature map. This bias helps adjust the activation level, providing additional flexibility for the network to better fit the data.

Summmary

1. **Activation Function:** Following the addition of the bias, the feature map values are typically passed through an activation function, such as ReLU (Rectified Linear Unit). The ReLU function introduces non-linearity by setting negative values to zero while retaining positive values, enabling the network

to learn more complex patterns.

Pooling

Pooling

- A **pooling layer** exhibits similarities to a **convolutional layer**.
  - Each neuron in a pooling layer is connected to a set of neurons within a **receptive field**.
- However, unlike convolutional layers, pooling layers do not possess **weights**.
  - Instead, they produce an output by applying an aggregating function, commonly **max** or **mean**.

Similar to convolutional layers, pooling layers allow specification of the receptive field size, padding, and stride. For the `MaxPool2D` function, the default receptive field size is 2×2.

In a pooling layer, specifically max pooling, the max function is inherently nondifferentiable because it involves selecting the maximum value from a set of inputs. However, in the context of backpropagation in neural networks, we can work around this by using a concept known as the “gradient of the max function.”

Here’s how it is done:

1. **Forward Pass**: During the forward pass, the max pooling layer selects the maximum value from each pooling region (e.g., a 2x2 window) and passes these values to the next layer.
2. **Backward Pass**: During backpropagation, the gradient is propagated only to the input that was the maximum value in the forward pass. This means that the derivative is 1 for the position that held the maximum value and 0 for all other positions within the pooling window.

This approach effectively allows the max operation to participate in gradient-based optimization processes like backpropagation, even though the max function itself is nondifferentiable. By assigning the gradient to the position of the maximum value, the network can learn which features are most important for the task at hand.

Pooling

- This subsampling process leads to a **reduction in network size**; each window of dimensions $f_h \times f_w$ is condensed to a single value, typically the **maximum** or **mean** of that window.
- A max pooling layer provides a degree of **invariance to small translations** (Géron (2019), § 14).

Pooling

1. **Dimensionality Reduction:** Pooling layers reduce the spatial dimensions (width and height) of the input feature maps. This reduction decreases the number of parameters and computational load in the network, which can help prevent overfitting.

Pooling

1. **Feature Extraction:** By summarizing the presence of features in a region, pooling layers help retain the most critical information while discarding less important details. This process enables the network to focus on the most salient features.

Pooling

1. **Translation Invariance:** Pooling introduces a degree of invariance to translations and distortions in the input. For instance, max pooling captures the most prominent feature in a local region, making the network less sensitive to small shifts or variations in the input.

Pooling

1. **Noise Reduction:** Pooling can help smooth out noise in the input by aggregating information over a region, thus emphasizing consistent features over random variations.

Pooling

1. **Hierarchical Feature Learning:** By reducing the spatial dimensions progressively through the layers, pooling layers allow the network to build a hierarchical representation of the input data, capturing increasingly abstract

and complex features at deeper layers.

Keras

In [15]:
```
import tensorflow as tf
from functools import partial

DefaultConv2D = partial(tf.keras.layers.Conv2D, kernel_size=3, padding

model = tf.keras.Sequential([
  DefaultConv2D(filters=64, kernel_size=7, input_shape=[28, 28, 1]),
  tf.keras.layers.MaxPool2D(),
  DefaultConv2D(filters=128),
  DefaultConv2D(filters=128),
  tf.keras.layers.MaxPool2D(),
  DefaultConv2D(filters=256),
  DefaultConv2D(filters=256),
  tf.keras.layers.MaxPool2D(),
  tf.keras.layers.Flatten(),
  tf.keras.layers.Dense(units=128, activation="relu", kernel_initializ
  tf.keras.layers.Dropout(0.5),
  tf.keras.layers.Dense(units=64, activation="relu", kernel_initialize
  tf.keras.layers.Dropout(0.5),
  tf.keras.layers.Dense(units=10, activation="softmax") ])

model.summary()
```

```
/Users/turcotte/opt/micromamba/envs/ml4bio/lib/python3.10/site-package
s/keras/src/layers/convolutional/base_conv.py:107: UserWarning:

Do not pass an `input_shape`/`input_dim` argument to a layer. When usin
g Sequential models, prefer using an `Input(shape)` object as the first
layer in the model instead.
```

Model: "sequential"

Layer (type)	Output Shape	
conv2d (Conv2D)	(None, 28, 28, 64)	
max_pooling2d (MaxPooling2D)	(None, 14, 14, 64)	
conv2d_1 (Conv2D)	(None, 14, 14, 128)	
conv2d_2 (Conv2D)	(None, 14, 14, 128)	
max_pooling2d_1 (MaxPooling2D)	(None, 7, 7, 128)	
conv2d_3 (Conv2D)	(None, 7, 7, 256)	
conv2d_4 (Conv2D)	(None, 7, 7, 256)	
max_pooling2d_2 (MaxPooling2D)	(None, 3, 3, 256)	
flatten (Flatten)	(None, 2304)	
dense (Dense)	(None, 128)	
dropout (Dropout)	(None, 128)	
dense_1 (Dense)	(None, 64)	
dropout_1 (Dropout)	(None, 64)	
dense_2 (Dense)	(None, 10)	

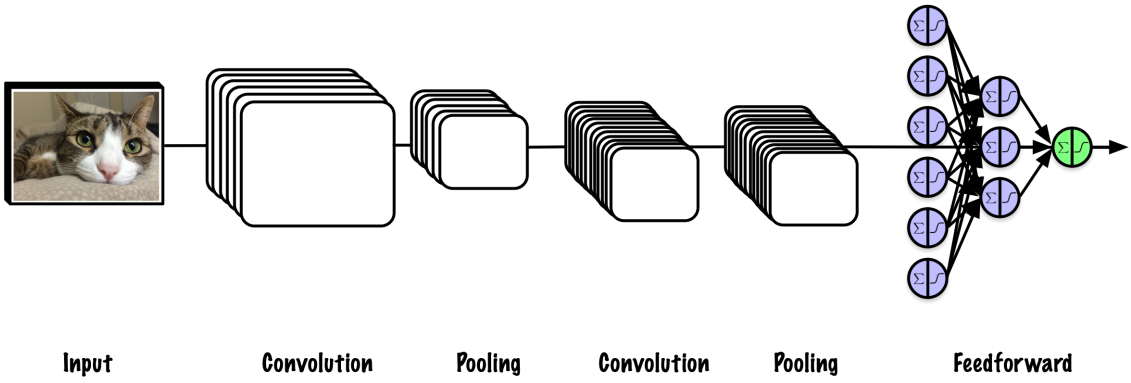
Total params: 1,413,834 (5.39 MB)
Trainable params: 1,413,834 (5.39 MB)
Non-trainable params: 0 (0.00 B)

Géron (2022) Chapter 11, test accuracy of 92% on the Fashion-MNIST dataset

The previously discussed model, which comprised fully connected (Dense) layers, attained a test accuracy of 88%.

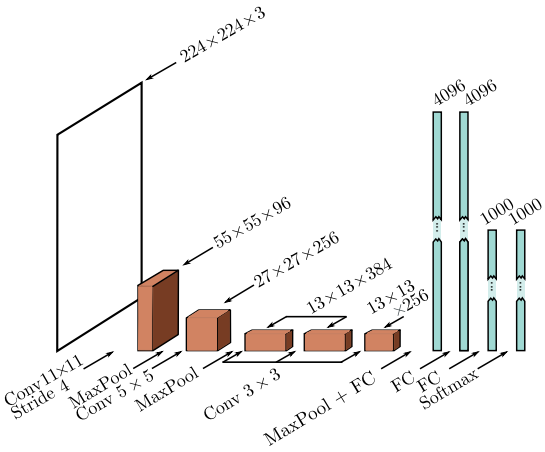
We will look at pooling next.

Convolutional Neural Networks



The architecture involves sequentially stacking several convolutional layers, each followed by a ReLU activation layer, and then a pooling layer. As this process continues, the spatial dimensions of the image representation decrease. Concurrently, the number of feature maps increases, as illustrated in our Keras example. At the top of this stack, a standard feedforward neural network is incorporated.

AlexNet

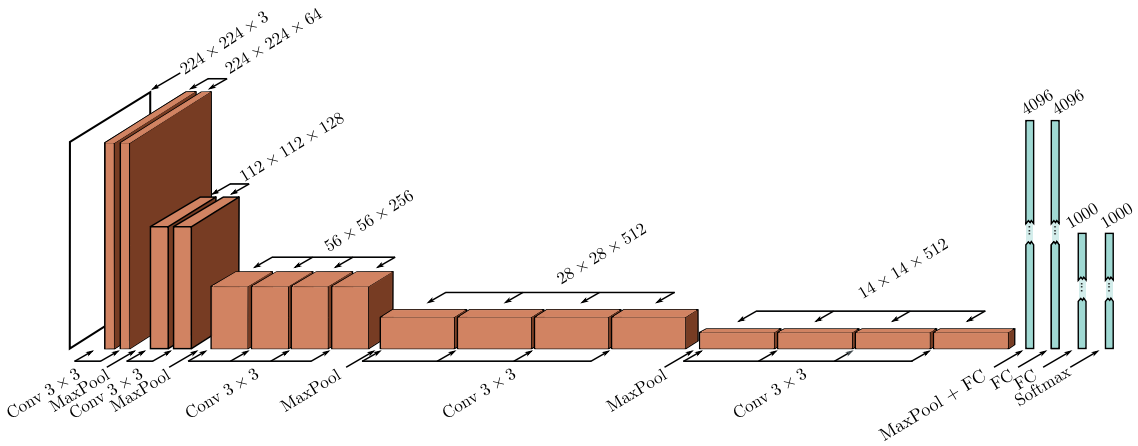


Krizhevsky, Sutskever, and Hinton (2012)

Attribution: Prince (2023)

AlexNet consists of eight layers with learnable parameters: five convolutional layers followed by three fully connected layers. The architecture also includes max-pooling layers, ReLU activation functions, and dropout to improve training performance and reduce overfitting.

VGG



Simonyan and Zisserman (2015)

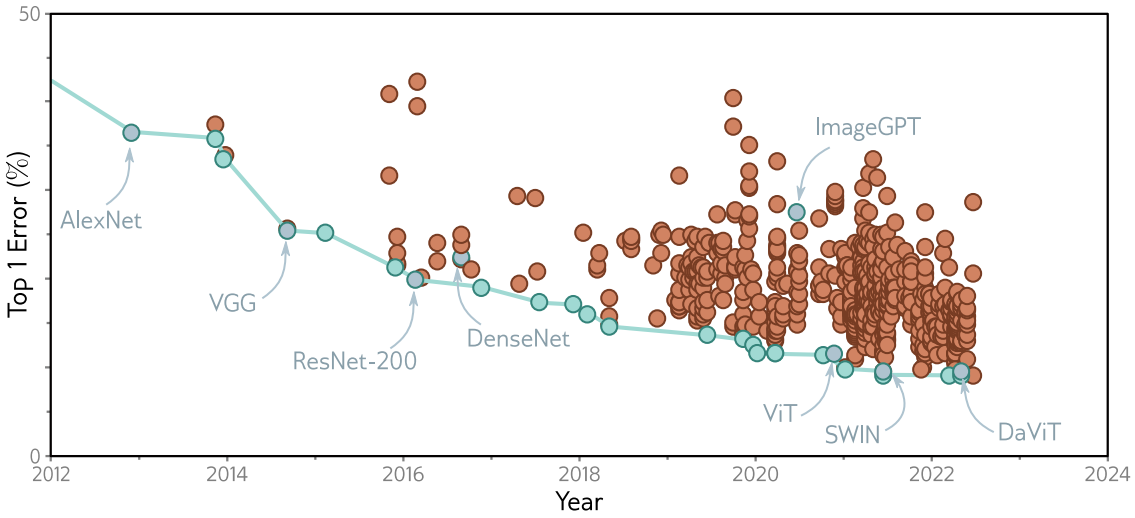
Attribution: Prince (2023)

Complementary information can be found here.

> Convolutional networks (ConvNets) currently set the state of the art in visual recognition. The aim of this project is to investigate how the ConvNet depth affects their accuracy in the large-scale image recognition setting.

> Our main contribution is a rigorous evaluation of networks of increasing depth, which shows that a significant improvement on the prior-art configurations can be achieved by increasing the depth to 16-19 weight layers, which is substantially deeper than what has been used in the prior art. To reduce the number of parameters in such very deep networks, we use very small 3×3 filters in all convolutional layers (the convolution stride is set to 1). Please see our publication for more details.

ConvNets Performance



Attribution: Prince (2023)

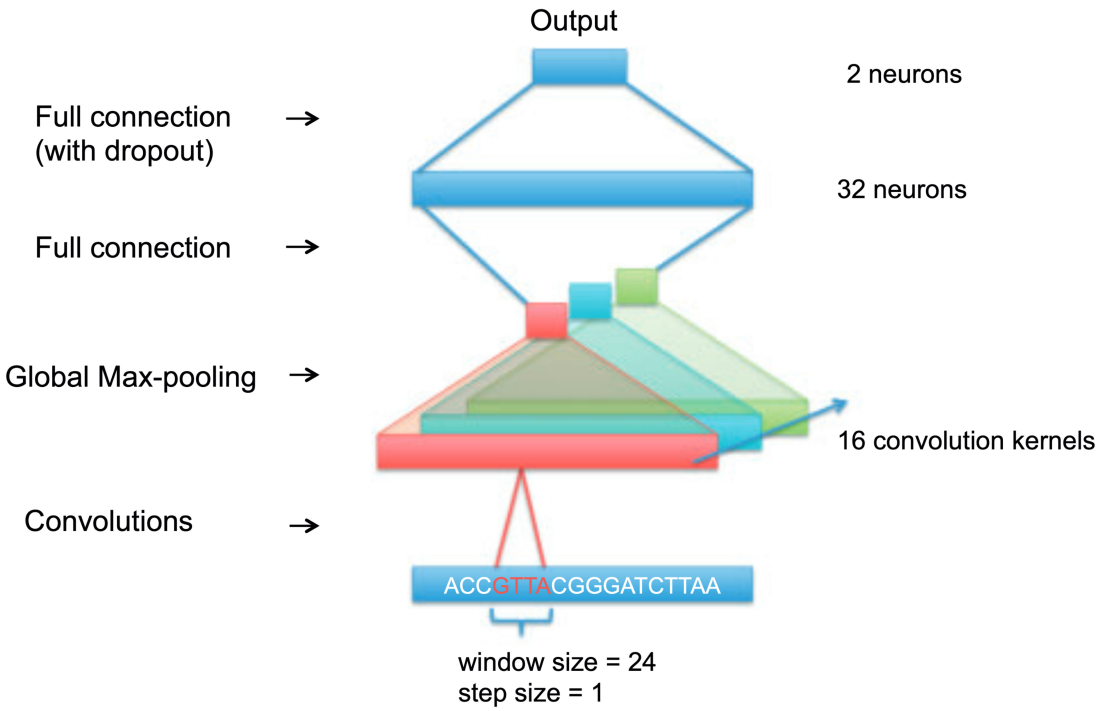
Final Word

As you might expect, the **number of layers** and **filters** are hyperparameters that are optimized through the process of **hyperparameter tuning**.

Attribution: @stefaan_cotteni

Case Study

Conv1D

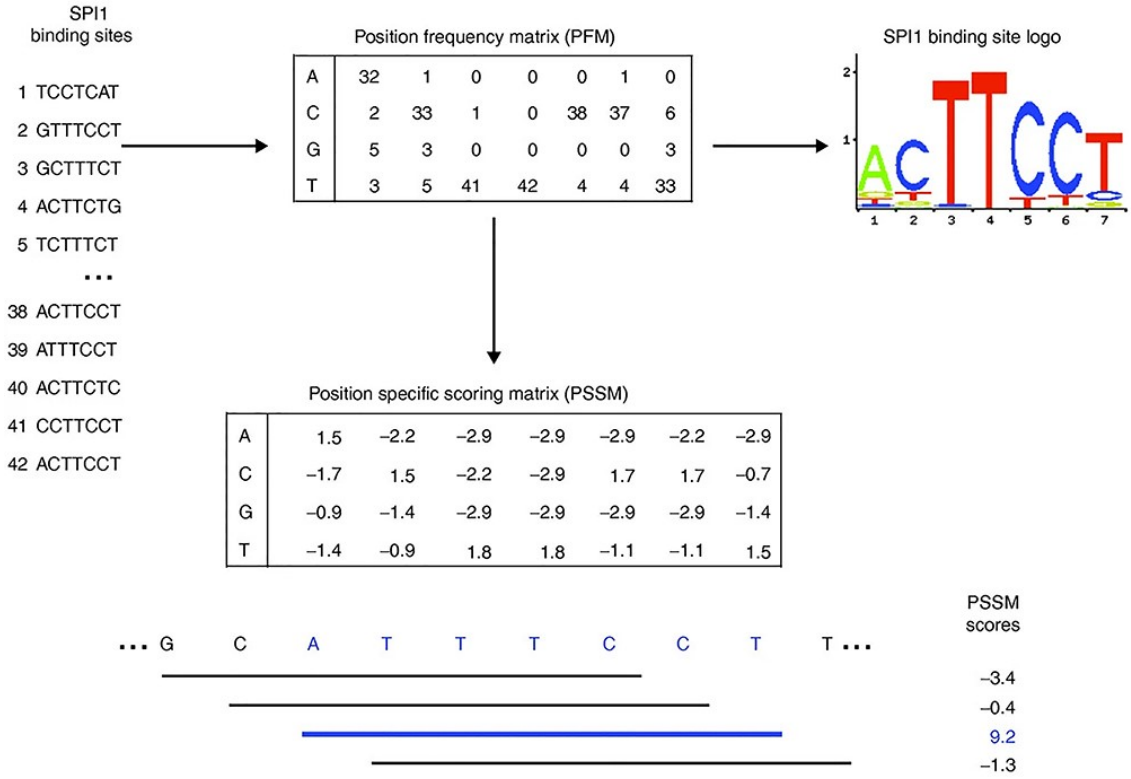


Zeng et al. (2016)

This study examines nine architectural variants of neural networks by altering their width, depth, and pooling configurations. The networks' performance is assessed across 690 distinct ChIP-seq experiments.

Conclusion: "We found for both tasks, classification performance increases with the number of convolution kernels, and the use of local pooling or more convolutional layers has little, if not negative, effect on the performance"

PSSM



He et al. (2020)

Primer on Deep Learning in Genomics

Imports and variable definitions.

In [16]:
```
import numpy as np
import pandas as pd

import matplotlib.pyplot as plt
import seaborn as sns

import requests

from sklearn.preprocessing import LabelEncoder, OneHotEncoder

np.set_printoptions(threshold=40)

# Define the base URL for accessing genomic sequence data

BASE_URL = 'https://raw.githubusercontent.com/abidlabs/deep-learning-g
SEQUENCES_URL = BASE_URL + 'sequences.txt'
```

Reproduced from Zou et al. (2019) and its accompanying Google Colab Tutorial.

Downloading Sequences

In [17]:
```
try:
    # Fetch the sequences from the URL
    response = requests.get(SEQUENCES_URL)
    response.raise_for_status()  # Raise an error if the request was u

    # Split the text data into a list of sequences, removing any empty
    sequences = response.text.split('\n')
    sequences = list(filter(None, sequences))  # Removes empty strings

    # Convert sequences into a pandas DataFrame for easy viewing
    df = pd.DataFrame(sequences, index=np.arange(1, len(sequences) + 1

    # Display the first few rows of the dataset
    print(df.head())

except requests.RequestException as e:
    print(f"Error fetching sequences: {e}")
```

	Sequences
1	CCGAGGGCTATGGTTTGGAAGTTAGAACCCTGGGGCTTCTCGCGGA...
2	GAGTTTATATGGCGCGAGCCTAGTGGTTTTTGTACTTGTTTGTCGC...
3	GATCAGTAGGGAAACAAACAGAGGGCCCAGCCACATCTAGCAGGTA...
4	GTCCACGACCGAACTCCCACCTTGACCGCAGAGGTACCACCAGAGC...
5	GGCGACCGAACTCCAACTAGAACCTGCATAACTGGCCTGGGAGATA...

Encoding

In [18]:
```
# Define fixed DNA bases
bases = ['A', 'C', 'G', 'T']

# Pre-fit the LabelEncoder with fixed bases
integer_encoder = LabelEncoder()
integer_encoder.fit(bases)

# Use the fixed integer mapping to define OneHotEncoder categories
categories = [integer_encoder.transform(bases)]
one_hot_encoder = OneHotEncoder(categories=categories)
```

Encoding

In [19]:
```
input_features = []

for sequence in sequences:
  integer_encoded = integer_encoder.fit_transform(list(sequence))
  integer_encoded = np.array(integer_encoded).reshape(-1, 1)
  one_hot_encoded = one_hot_encoder.fit_transform(integer_encoded)
  input_features.append(one_hot_encoded.toarray())

X = np.stack(input_features)
```

```
print("Example sequence\n----------------------")
print('DNA Sequence #1:\n',sequences[0][:10],'...',sequences[0][-10:])
print('One hot encoding of Sequence #1:\n',X[0].T)
```

```
Example sequence
----------------------
DNA Sequence #1:
 CCGAGGGCTA ... CGCGGACACC
One hot encoding of Sequence #1:
 [[0. 0. 0. ... 1. 0. 0.]
 [1. 1. 0. ... 0. 1. 1.]
 [0. 0. 1. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]]
```

Downloading Responses

In [20]:
```
LABELS_URL = BASE_URL + 'labels.txt'

labels = requests.get(LABELS_URL).text.split('\n')
labels = list(filter(None, labels))  # removes empty sequences
labels = np.array(labels).reshape(-1, 1)

one_hot_encoder = OneHotEncoder(categories='auto')

y = one_hot_encoder.fit_transform(labels).toarray()

print('Labels:\n', labels.T)
print('One-hot encoded labels:\n', y.T)
```

```
Labels:
 [['0' '0' '0' ... '0' '1' '1']]
One-hot encoded labels:
 [[1. 1. 1. ... 1. 0. 0.]
 [0. 0. 0. ... 0. 1. 1.]]
```

Train & Test Sets

In [21]:
```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.
```

Model

In [22]:
```
from tensorflow.keras.layers import Conv1D, Dense, MaxPooling1D, Flatt
from tensorflow.keras.models import Sequential

model = Sequential()
```

```
model.add(Conv1D(filters=32, kernel_size=12, input_shape=(X_train.shap
model.add(MaxPooling1D(pool_size=4))
model.add(Flatten())
model.add(Dense(16, activation='relu'))
model.add(Dense(2, activation='softmax'))

model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['

model.summary()
```

```
/Users/turcotte/opt/micromamba/envs/ml4bio/lib/python3.10/site-package
s/keras/src/layers/convolutional/base_conv.py:107: UserWarning:

Do not pass an `input_shape`/`input_dim` argument to a layer. When usin
g Sequential models, prefer using an `Input(shape)` object as the first
layer in the model instead.
```

Model: "sequential_1"

Layer (type)	Output Shape	
conv1d (Conv1D)	(None, 39, 32)	
max_pooling1d (MaxPooling1D)	(None, 9, 32)	
flatten_1 (Flatten)	(None, 288)	
dense_3 (Dense)	(None, 16)	
dense_4 (Dense)	(None, 2)	

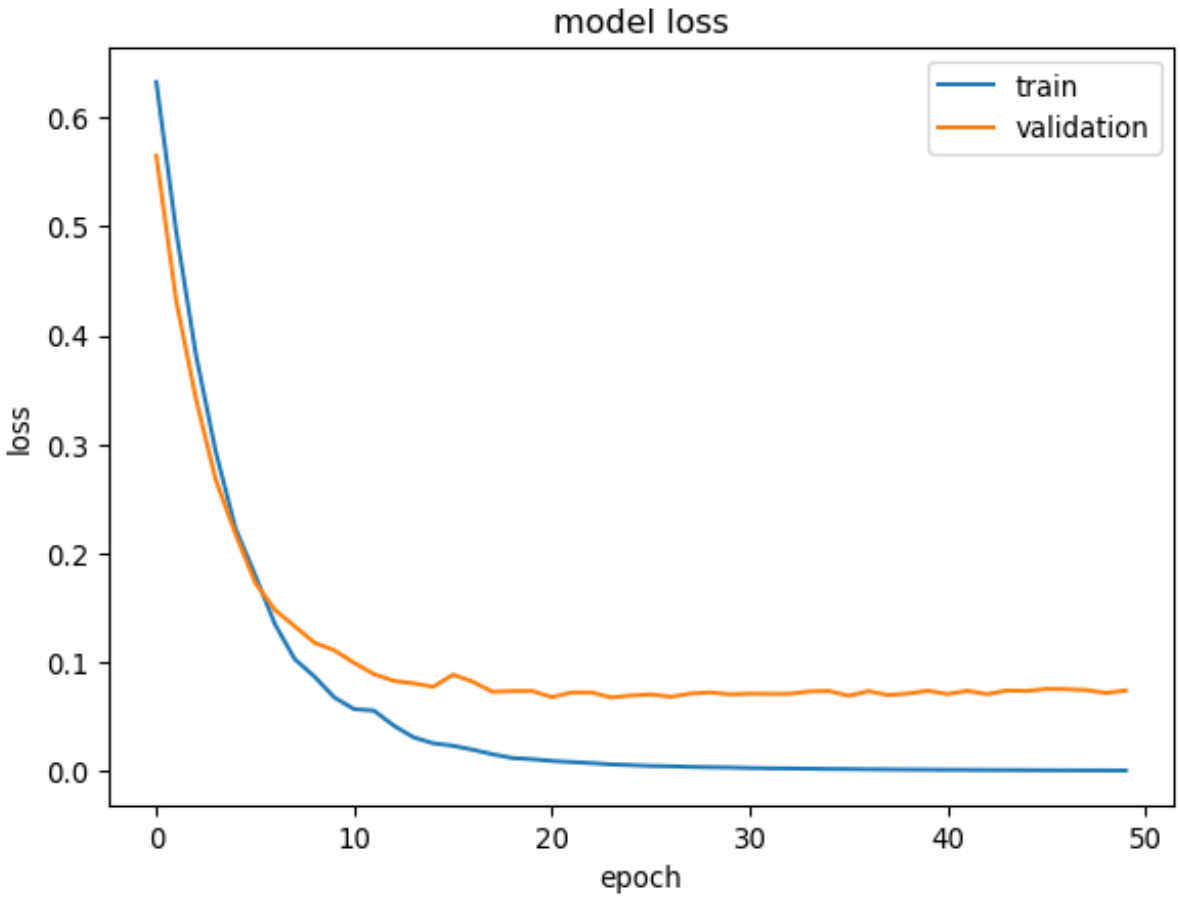
Total params: 6,226 (24.32 KB)

Trainable params: 6,226 (24.32 KB)

Non-trainable params: 0 (0.00 B)

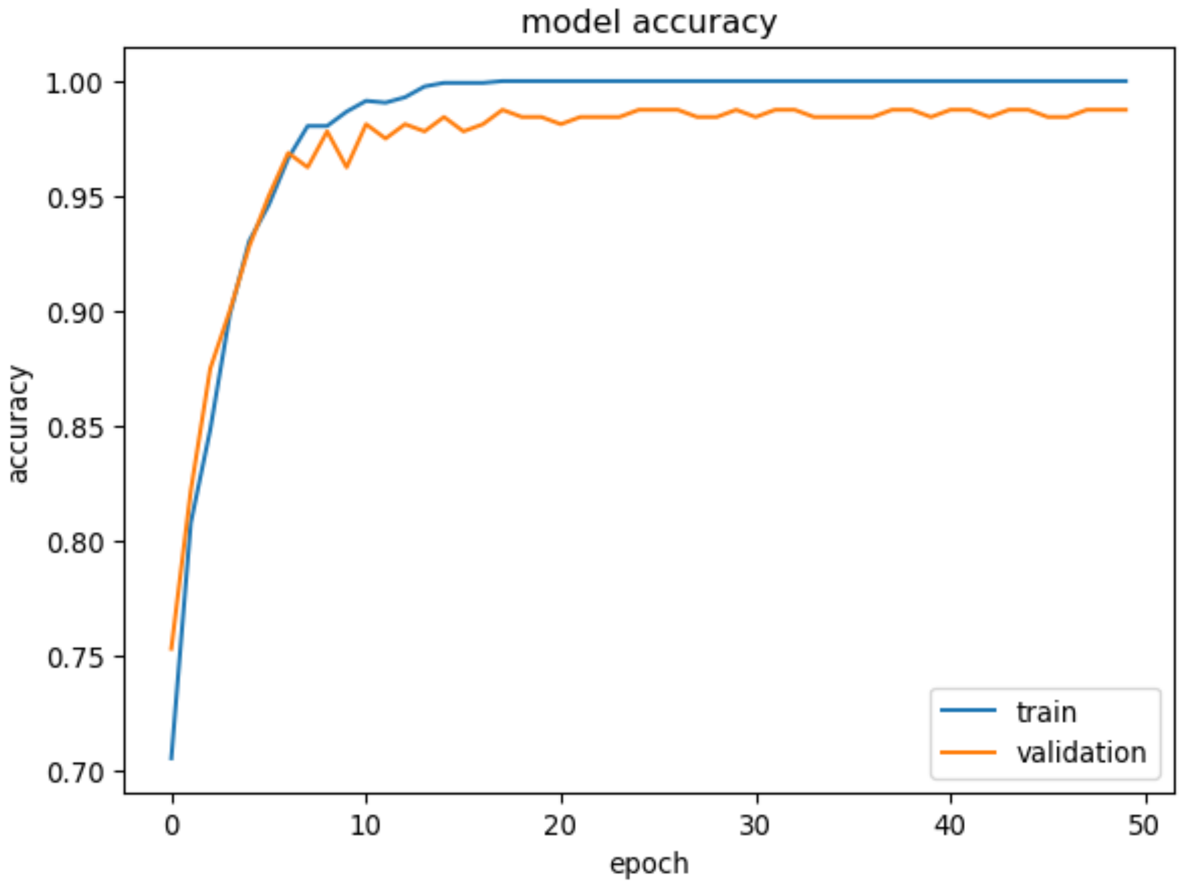
Training

In [23]:
```
history = model.fit(X_train, y_train, epochs=50, verbose=0, validation

plt.figure()
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'validation'])
plt.show()
```



Training and Validation Accuracy

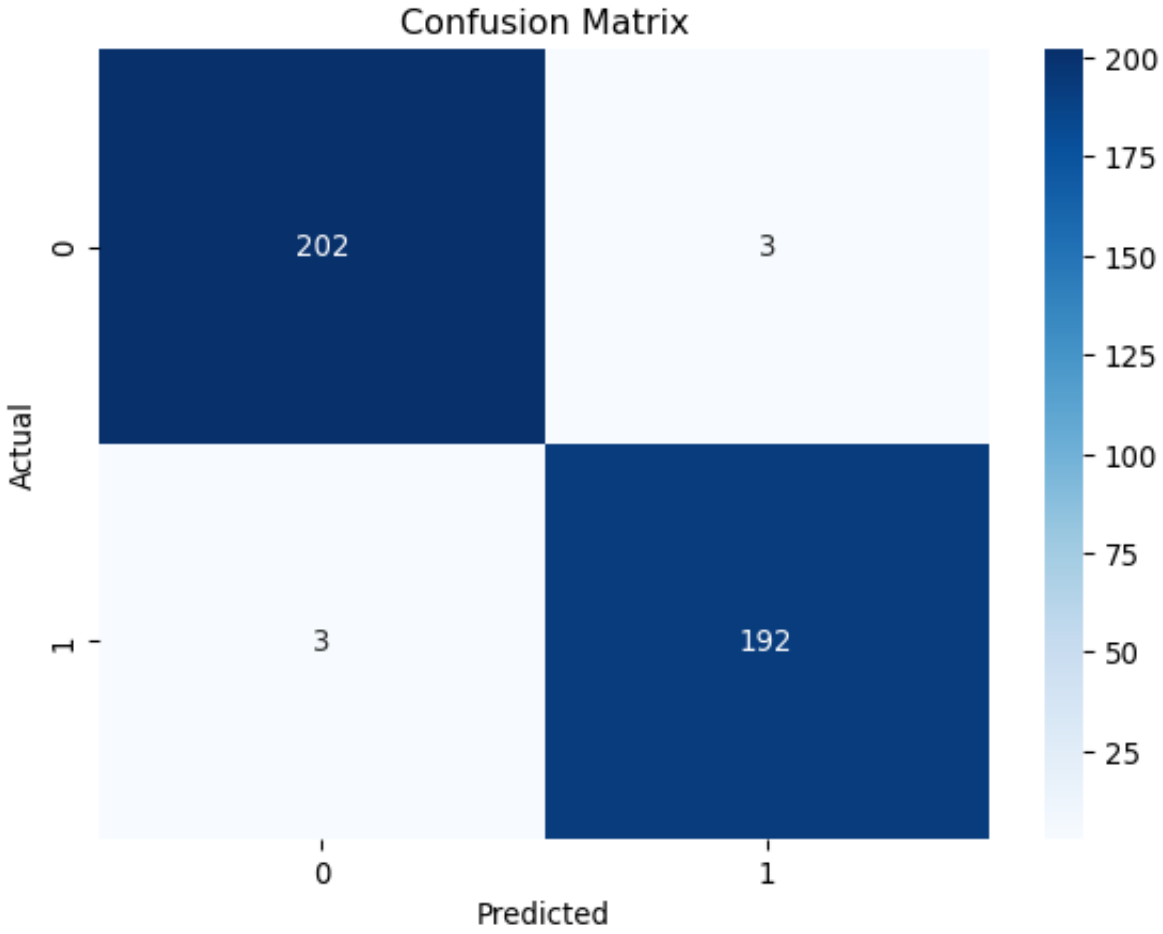
In [24]:
```
plt.figure()
plt.plot(history.history['binary_accuracy'])
plt.plot(history.history['val_binary_accuracy'])
plt.title('model accuracy')
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['train', 'validation'])
plt.show()
```



Performance on Test Set

In [25]:
```
from sklearn.metrics import confusion_matrix

y_pred = model.predict(np.stack(X_test))

cm = confusion_matrix(np.argmax(y_test, axis=1),
                      np.argmax(y_pred, axis=1))

sns.heatmap(cm, annot=True, fmt='d', cmap='Blues')
plt.title(f'Confusion Matrix')
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.show()
```

```
1/13 ━━━━━━━━━━━━━━━━━━━━ 0s 27ms/step
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step
```



Prologue

Summary

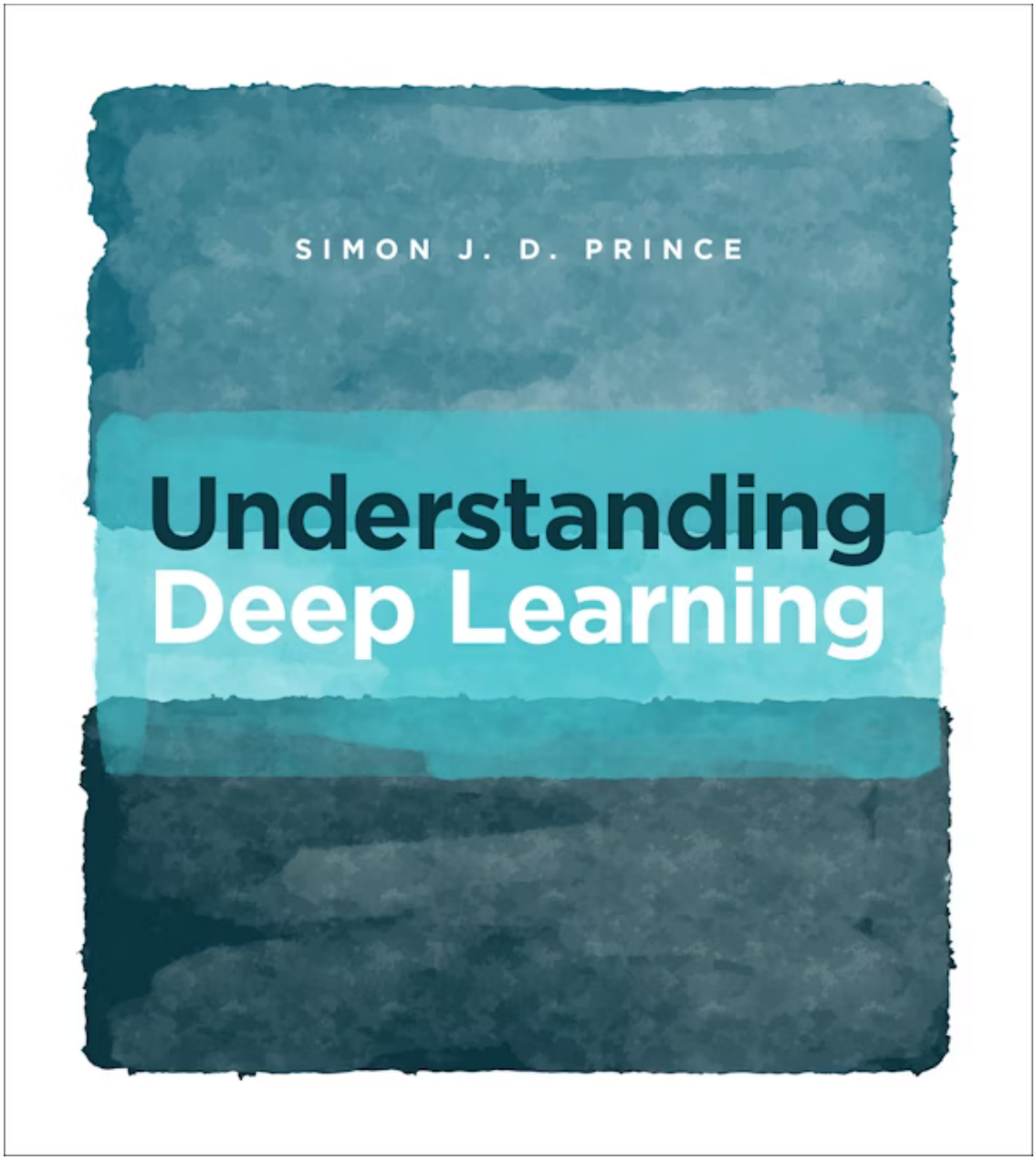
Throughout this lecture, we examined the key principles and operations of Convolutional Neural Networks (CNNs):

1. **Hierarchy of Concepts in Deep Learning**
   - Deep learning models construct hierarchical representations of data, reducing reliance on manual feature engineering.
   - Deeper networks achieve greater parameter efficiency compared to shallow networks.
2. **Convolutional Neural Networks (CNNs)**
   - CNNs specialize in processing grid-structured data, such as images and genomic sequences.
   - Weight sharing and local connectivity reduce the number of parameters compared to fully connected networks.
3. **Kernels and Convolution Operations**

- Kernels extract local features by sliding over input data and performing element-wise multiplication.
- The output feature maps highlight important patterns in the data.

4. **Receptive Field, Padding, and Stride**
   - The receptive field defines the portion of input visible to a neuron in a convolutional layer.
   - Padding helps maintain spatial dimensions, while stride controls the step size of kernel movement.
5. **Filters and Feature Maps**
   - Filters are automatically learned during training to capture meaningful patterns in data.
   - Shared parameters across neurons enhance efficiency and generalization.
6. **Convolutional Layers**
   - Layers apply convolution operations followed by an activation function like ReLU.
   - Non-linearity introduced by ReLU enables learning of complex representations.
7. **Pooling Layers**
   - Pooling reduces spatial dimensions and computational costs while preserving key features.
   - Max pooling provides translation invariance, improving robustness to variations in input.
8. **CNN Architectures and Applications**
   - Stacking convolutional and pooling layers leads to feature-rich representations.
   - CNNs are widely used in image processing, bioinformatics, and medical diagnostics.
9. **Hyperparameter Tuning**
   - Key parameters include filter size, depth, stride, and padding.
   - Regularization techniques, such as dropout, prevent overfitting.

Further Reading



- Understanding Deep Learning (Prince 2023) is a recently published textbook focused on the foundational concepts of deep learning.
- It begins with fundamental principles and extends to contemporary topics such as transformers, diffusion models, graph neural networks, autoencoders, adversarial networks, and reinforcement learning.
- The textbook aims to help readers comprehend these concepts without delving excessively into theoretical details.
- It includes sixty-eight Python notebook exercises.
- The book follows a “read-first, pay-later” model.

Resources

- **A guide to convolution arithmetic for deep learning**
- Authors: Vincent Dumoulin and Francesco Visin
- Last revised: 11 Jan 2018
  - arXiv:1603.07285
  - GitHub Repository

Next lecture

- Student presentations!

References

Consens, Micaela E., Cameron Dufault, Michael Wainberg, Duncan Forster, Mehran Karimzadeh, Hani Goodarzi, Fabian J. Theis, Alan Moses, and Bo Wang. 2025. "Transformers and genome language models." *Nature Machine Intelligence*, 1–17. https://doi.org/10.1038/s42256-025-01007-9.

Géron, Aurélien. 2019. *Hands-on Machine Learning with Scikit-Learn, Keras, and TensorFlow*. 2nd ed. O'Reilly Media.

———. 2022. *Hands-on Machine Learning with Scikit-Learn, Keras, and TensorFlow*. 3rd ed. O'Reilly Media, Inc.

Goodfellow, Ian, Yoshua Bengio, and Aaron Courville. 2016. *Deep Learning*. Adaptive Computation and Machine Learning. MIT Press. https://dblp.org/rec/books/daglib/0040158.

He, Ying, Zhen Shen, Qinhu Zhang, Siguo Wang, and De-Shuang Huang. 2020. "A survey on deep learning in DNA/RNA motif mining." *Briefings in Bioinformatics* 22 (4): bbaa229. https://doi.org/10.1093/bib/bbaa229.

Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E Hinton. 2012. "ImageNet Classification with Deep Convolutional Neural Networks." In *Advances in Neural Information Processing Systems*, edited by F. Pereira, C. J. Burges, L. Bottou, and K. Q. Weinberger. Vol. 25. Curran Associates, Inc. https://proceedings.neurips.cc/paper_files/paper/2012/file/c399862d3b9d6b76c843
Paper.pdf.

LeCun, Yann, Yoshua Bengio, and Geoffrey Hinton. 2015. "Deep Learning." *Nature* 521 (7553): 436–44. https://doi.org/10.1038/nature14539.

Lecun, Y., L. Bottou, Y. Bengio, and P. Haffner. 1998. “Gradient-Based Learning Applied to Document Recognition.” *Proceedings of the IEEE* 86 (11): 2278–2324. https://doi.org/10.1109/5.726791.

Prince, Simon J. D. 2023. *Understanding Deep Learning*. The MIT Press. http://udlbook.com.

Simonyan, Karen, and Andrew Zisserman. 2015. “Very Deep Convolutional Networks for Large-Scale Image Recognition.” In *International Conference on Learning Representations*.

Zeng, Haoyang, Matthew D Edwards, Ge Liu, and David K Gifford. 2016. “Convolutional Neural Network Architectures for Predicting DNA-Protein Binding.” *Bioinformatics (Oxford, England)* 32 (12): i121–27. https://doi.org/10.1093/bioinformatics/btw255.

Zou, James, Mikael Huss, Abubakar Abid, Pejman Mohammadi, Ali Torkamani, and Amalio Telenti. 2019. “A Primer on Deep Learning in Genomics.” *Nature Genetics* 51 (1): 12–18. https://doi.org/10.1038/s41588-018-0295-5.

Marcel **Turcotte**

Marcel.Turcotte@uOttawa.ca

School of Electrical Engineering and **Computer Science** (EE**CS**)

University of Ottawa