CE6146
Introduction to Deep Learning
Convolutional Neural Networks

Chia-Ru Chung

Department of Computer Science and Information Engineering

National Central University

2023/10/12

20231005 Exercise

1. D	2. C	3. A	4. D	5. B
6. A	7. B	8. C	9. D	10. A
11. C	12. C	13. C	14. C	15. C
16. B	17. D	18. B	19. B	20. C

Outline

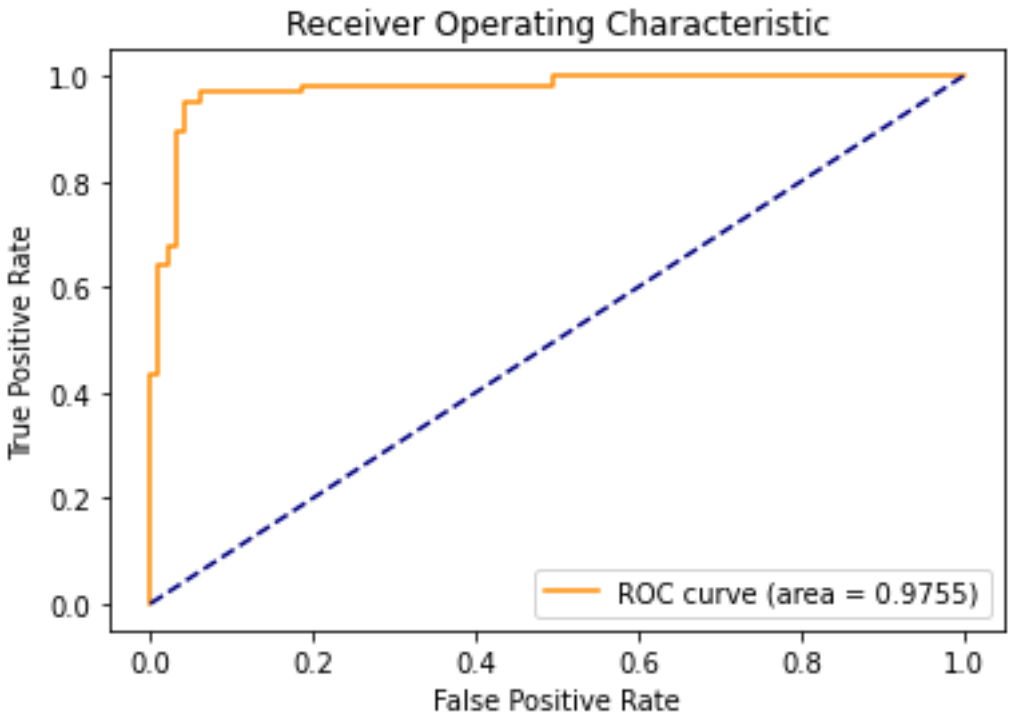
- Review
- Convolutional Neural Networks
- Hand-on Convolutional Neural Networks

Review

- ROC and PR Curve
- Example for Forward and Backward Pass
- Gradient Decent

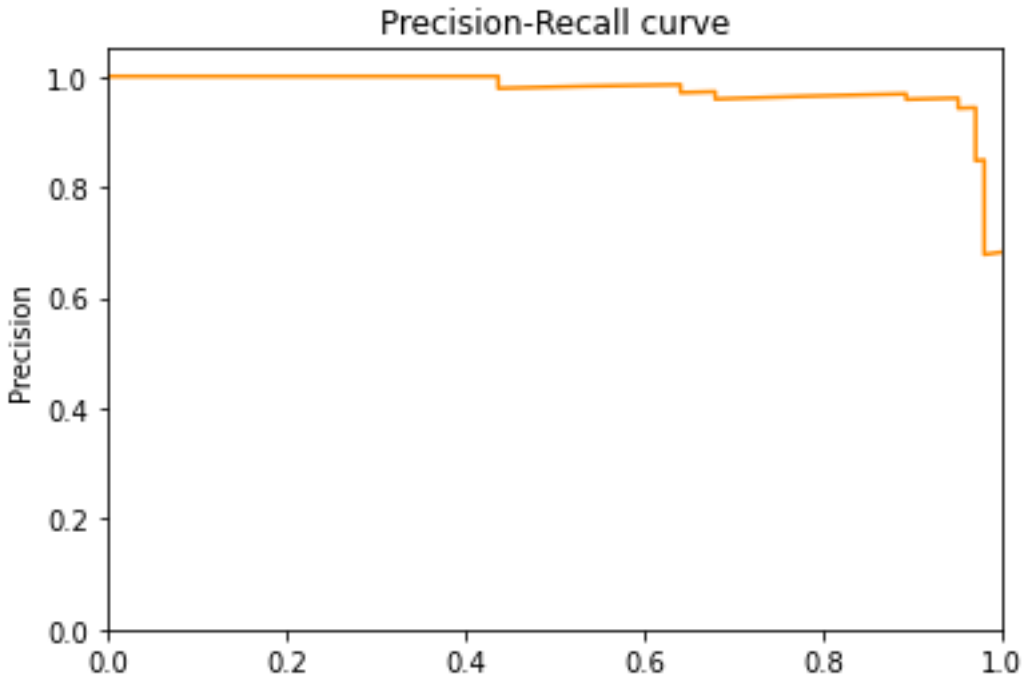
Example for ROC

```python
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import roc_curve, auc
import matplotlib.pyplot as plt

# Step 1: Generate a dataset
X, y = make_classification(n_samples=1000, n_features=10, n_classes=2, random_state=0)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=0)

# Step 2: Train a Logistic Regression model
clf = LogisticRegression()
clf.fit(X_train, y_train)

# Step 3: Get predicted probabilities
y_pred_prob = clf.predict_proba(X_test)[:, 1]

# Step 4: Calculate TPR and FPR
fpr, tpr, thresholds = roc_curve(y_test, y_pred_prob)
roc_auc = round(auc(fpr, tpr),4)

# Step 5: Plot the ROC Curve
plt.figure()
plt.plot(fpr, tpr, color='darkorange', label=f'ROC curve (area = {roc_auc})')
plt.plot([0, 1], [0, 1], color='navy', linestyle='--')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver Operating Characteristic')
plt.legend(loc="lower right")
plt.show()
```



Example for PR Curves

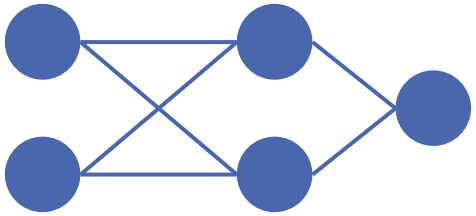
```python
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import precision_recall_curve, average_precision_score
import matplotlib.pyplot as plt

# Step 1: Generate synthetic data
X, y = make_classification(n_samples=1000, n_features=10, n_classes=2, random_state=0)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=0)

# Step 2: Train model
clf = LogisticRegression()
clf.fit(X_train, y_train)

# Step 3: Predict probabilities
y_pred_prob = clf.predict_proba(X_test)[:, 1]

# Step 4: Compute Precision and Recall
precision, recall, thresholds = precision_recall_curve(y_test, y_pred_prob)
average_precision = average_precision_score(y_test, y_pred_prob)

# Step 5: Plot PR Curve
plt.plot(recall, precision, color='darkorange')
plt.ylabel('Precision')
plt.ylim([0.0, 1.05])
plt.xlim([0.0, 1.0])
plt.title('Precision-Recall curve')
plt.show()
```



Example for Forward and Backward Pass

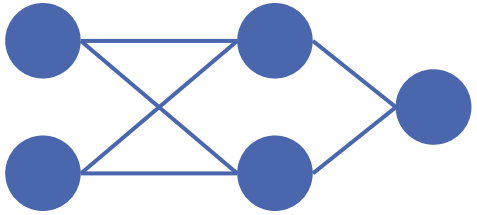
- Let's develop a simple feedforward neural network to predict height based on age and weight. The network architecture will consist of:
  - Input Layer: 2 neurons (for age and weight)
  - One Hidden Layer: 2 neurons (ReLU activation)
  - Output Layer: 1 neuron (Linear activation for regression)

```
# Initialize the dataset (10 data points)
# Columns: [Age, Weight, Height]
data = np.array([
    [25, 70, 175],
    [30, 80, 180],
    [35, 85, 178],
    [40, 88, 176],
    [45, 70, 170],
    [50, 75, 169],
    [55, 77, 165],
    [60, 88, 160],
    [65, 70, 158],
    [70, 72, 155]
])
```



Forward Pass

```
# Initialize weights and biases
input_neurons = 2
hidden_neurons = 2
output_neurons = 1

W1 = np.random.uniform(size=(input_neurons, hidden_neurons))
b1 = np.random.uniform(size=(1, hidden_neurons))
W2 = np.random.uniform(size=(hidden_neurons, output_neurons))
b2 = np.random.uniform(size=(1, output_neurons))

# Learning rate
learning_rate = 0.01
```



```
    # Forward Pass
    Z1 = np.dot(X, W1) + b1
    A1 = relu(Z1)
    Z2 = np.dot(A1, W2) + b2
    A2 = Z2  # Linear activation for regression

    # Loss Computation (Mean Squared Error)
    loss = np.mean((y - A2)**2)
```

Backward Pass

```
# Backward Pass
dA2 = 2 * (A2 - y)
dZ2 = dA2
dW2 = np.dot(A1.T, dZ2)
db2 = np.sum(dZ2, axis=0, keepdims=True)

dA1 = np.dot(dZ2, W2.T)
dZ1 = dA1 * relu_derivative(Z1)
dW1 = np.dot(X.T, dZ1)
db1 = np.sum(dZ1, axis=0, keepdims=True)

# Update Weights and Biases
W1 -= learning_rate * dW1
b1 -= learning_rate * db1
W2 -= learning_rate * dW2
b2 -= learning_rate * db2
```

```
# Function to compute ReLU activation
def relu(x):
    return np.maximum(0, x)

# Function to compute the derivative of ReLU
def relu_derivative(x):
    return np.where(x > 0, 1, 0)
```

$$loss = (A_2 - y)^2$$
$$A_2 = Z_2 = W_2^T A_1 + b_2$$
$$A_1 = ReLu(Z_1)$$
$$Z_1 = W_1^T X + b_1$$

$$\frac{\partial loss}{\partial W_1} = \frac{\partial loss}{\partial Z_1} \times \frac{\partial Z_1}{\partial W_1},$$
$$\frac{\partial loss}{\partial b_1} = \sum_{i=1}^{n} \frac{\partial loss}{\partial Z_{1i}}$$
$$\frac{\partial loss}{\partial W_2} = \frac{\partial loss}{\partial Z_2} \times \frac{\partial Z_2}{\partial W_2}$$
$$\frac{\partial loss}{\partial b_2} = \sum_{i=1}^{n} \frac{\partial loss}{\partial Z_{2i}}$$

Adaptive Gradient Decent

- Adaptive gradient descent algorithms are variants of the standard gradient descent algorithm that adapt the learning rates during training.
- These methods modify the learning rate for each parameter based on the historical gradient information of that parameter.
- This can result in faster convergence and can also alleviate some issues like vanishing or exploding gradients.

Why Adaptive Gradient Descent Methods Can Offer Faster Convergence (1/3)

- Individual Learning Rates:
  - Unlike standard gradient descent, adaptive methods use an individual learning rate for each parameter.
  - This can be particularly beneficial for optimizing parameters that have infrequent but significant updates.

Why Adaptive Gradient Descent Methods Can Offer Faster Convergence (2/3)

- Automatic Adjustment:
  - The learning rates are adjusted automatically during training.
  - This eliminates the need to manually tune the learning rate, making it easier to manage and faster to converge.

Why Adaptive Gradient Descent Methods Can Offer Faster Convergence (3/3)

- Balancing Updates:
  - In high-dimensional spaces or in situations where some features are sparse, the adaptive learning rates help balance the rate of update between frequent and rare features, making the optimization process more efficient.

How They Alleviate Issues Like Vanishing or Exploding Gradients (1/3)

- Scaling Factor:
  - Adaptive methods scale the gradients by a factor that considers the historical gradient values.
  - This means that if a parameter receives high gradients, its updates will be scaled down, and vice versa.
  - This mechanism naturally helps in mitigating the exploding gradients problem.

How They Alleviate Issues Like Vanishing or Exploding Gradients (2/3)

- Stabilizing Updates:
  - By adapting the learning rates, these methods can provide a form of implicit regularization.
  - This helps in stabilizing the updates, making the network less susceptible to fluctuations that could lead to exploding gradients.

How They Alleviate Issues Like Vanishing or Exploding Gradients (3/3)

- Epsilon Term:
  - The addition of a small epsilon in the denominator also avoids division by zero and provides numerical stability.
  - This can be crucial for dealing with vanishing gradients, ensuring that the learning rates don’t skyrocket.

Adagrad (Adaptive Gradient Algorithm)

- Each parameter ϕ_i has its own learning rate, which is adjusted based on the sum of the squares of its past gradients.
- The more a parameter is updated, the smaller its learning rate becomes, allowing the algorithm to converge faster for frequent features while not neglecting rare features.

$$\phi_i^{(k+1)} \leftarrow \phi_i^{(k)} - \eta \cdot \frac{\partial L}{\partial \phi_i^{(k)}} \cdot \left[\epsilon + \sum_{m=0}^{k} \left(\frac{\partial L}{\partial \phi_i^{(m)}}\right)^2\right]^{-1/2}$$

- η: Global learning rate, a hyperparameter set before training
- ϵ: A small constant added for numerical stability, usually set to 1×10^{-8}

RMSprop (Root Mean Square Propagation)

- It improves upon Adagrad by using a moving average of the squared gradient.
- This ensures that the learning rate does not decrease too fast, which is especially useful in the case of non-convex optimization problems like neural networks.

$$\phi_i^{(k+1)} \leftarrow \phi_i^{(k)} - \eta \cdot \frac{\partial L}{\partial \phi_i^{(k)}} \cdot [\epsilon + s_i^{(k)}]^{-1/2}$$

$$s_i^{(k+1)} = \gamma s_i^{(k)} + (1-\gamma) \sum_{m=0}^{k} \left(\frac{\partial L}{\partial \phi_i^{(m)}} \right)^2$$

- γ: Decay factor, a hyperparameter usually set between 0.9 and 0.99. It determines the rate at which the moving average forgets past squared gradients.

Adam (Adaptive Moment Estimation)

- It combines RMSprop and momentum.
- It computes adaptive learning rates for each parameter and also keeps an estimate of the first moment (the mean) of the gradients..

$$\phi_i^{(k+1)} \leftarrow \phi_i^{(k)} - \eta \cdot m_i^{(k)} \cdot [\epsilon + v_i^{(k)}]^{-1/2}$$

$$m_i^{(k+1)} = \gamma_1 m_i^{(k)} + (1-\gamma_1)\frac{\partial L}{\partial \phi_i^{(k)}}$$

$$v_i^{(k+1)} = \gamma_2 v_i^{(k)} + (1-\gamma_2)\sum_{m=0}^{k}\left(\frac{\partial L}{\partial \phi_i^{(m)}}\right)^2$$

- m_i : First moment estimate (mean) of the gradient for ϕ_i
- v_i : Second moment estimate (uncentered variance) of the gradient for ϕ_i
- γ_1: Exponential decay rate for the first moment estimate, usually close to 1 (e.g., 0.9).
- γ_2 : Exponential decay rate for the second moment estimate, also usually close to 1 (e.g., 0.999).

Comparisons of Three Methods

- Adagrad is better suited for problems with sparse gradients.
- RMSprop is an extension of Adagrad that deals better with problems of diminishing learning rates.
- Adam combines the advantages of RMSprop and Momentum, and is often recommended for most deep learning tasks.

Why Adagrad Is Good for Sparse Gradients

- Adagrad adjusts the learning rate for each feature based on the historical gradients.
- Features that are sparse but informative will have their learning rates boosted, allowing the model to learn effectively from them.
- Example:

  Imagine a text classification problem where you have a very large vocabulary (say, 100,000 words). However, any specific text sample will contain only a small subset of the entire vocabulary. In this case, the gradient for frequent words should be updated more cautiously, whereas infrequent words need more aggressive updates for the model to learn anything useful from them.

Why RMSprop Is Good for Non-convex Problems

- RMSprop uses a moving average of squared gradients to normalize the gradient descent updates, which helps in navigating the non-convex loss surfaces effectively.
- Example:

  In neural networks, especially deep ones, the loss surface is non-convex. Standard optimization techniques can easily get stuck in local minima.

Why Adam Is Generally Recommended

- Adam not only adapts the learning rates but also uses a moving average of past gradients (momentum) to get a more reliable direction for parameter updates.
- Example:

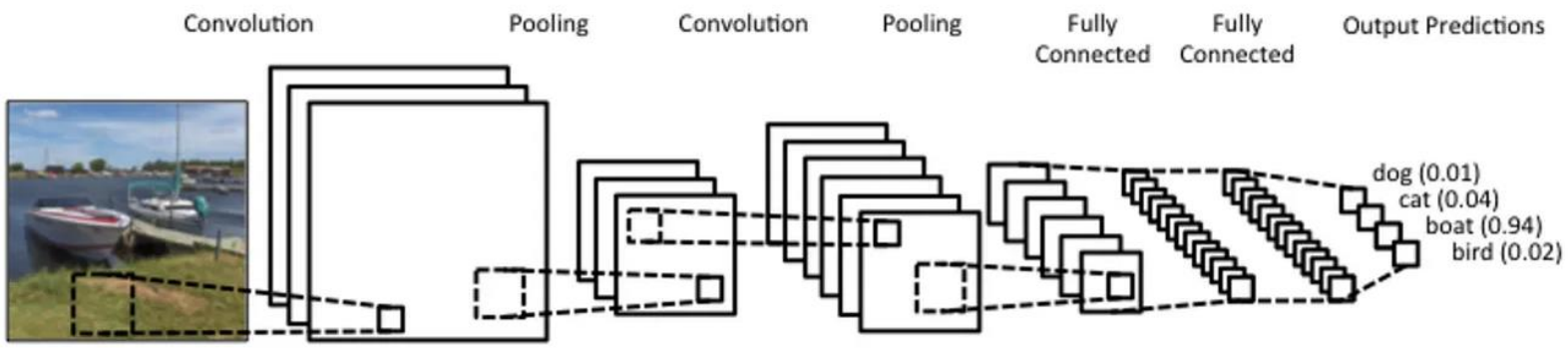
  In complex tasks like machine translation or image recognition, the optimization landscape is both non-convex and noisy. A method that can both adapt learning rates and smooth out noisy gradients can be very beneficial.

Convolutional Neural Networks

- Introduction
- Basics of Convolutional Layers
- Pooling Layers and Architecture

Source: https://medium.com/jameslearningnote/%E8%B3%87%E6%96%99%E5%88%86%E6%9E%90-%E6%A9%9F%E5%99%A8%E5%AD%B8%E7%BF%92-%E7%AC%AC5-1%E8%AC%9B-%E5%8D%B7%E7%A9%8D%E7%A5%9E%E7%B6%93%E7%B6%B2%E7%B5%A1%E4%BB%8B%E7%B4%B9-convolutional-neural-network-4f8249d65d4f

Introduction



- What are Convolutional Neural Networks?

  Convolutional Neural Networks (CNNs) are a specialized type of neural networks designed for processing data with a grid-like topology, such as an image.

- Layer Composition:

  A typical CNN is composed of a series of layers that transform the input image into an output. These layers often include convolutional layers, pooling layers, and fully connected layers.

- Role in Deep Learning:

  CNNs are one of the foundational architectures in deep learning, particularly for tasks related to image recognition.

Importance in Deep Learning

- Automated Feature Learning:

  Unlike traditional machine learning models, CNNs can automatically learn to identify important features without manual intervention.

- Efficiency:

  CNNs are highly efficient in terms of computational resources and offer advantages like parameter sharing.

- Scalability:

  CNNs can effectively handle large and high-dimensional data, making them ideal for tasks like image and video recognition.

- Versatility:

  Beyond image processing, CNN architectures have been adapted for various other domains including NLP, time-series analysis, and more.

Applications of CNNs in the Real World

- Image Classification:

  CNNs are the backbone of many image classification systems, identifying objects and features within images.

- Facial Recognition:

  Facial identification and verification systems in various security applications are often powered by CNNs.

- Medical Imaging:

  CNNs assist in medical diagnoses by analyzing X-rays, MRIs, and other imaging data.

- Autonomous Vehicles:

  Self-driving cars use CNNs for tasks such as object detection, lane tracking, and navigation.

- Video Surveillance:

  CNNs are used in real-time video analysis and surveillance systems for tasks like anomaly detection.

What is Convolution

- Convolution is an operation that takes an input image and a filter (also known as a kernel), and produces a feature map.
- The filter is a small matrix that slides over the input image to produce the feature map.
- Mathematically, convolution involves multiplying the filter values by the original pixel values in the image. These products are summed up, and the result forms a single pixel in the output feature map.

Why Convolution

- Convolution helps the model focus on local spatial regions of the input.
- The operation detects basic features like edges in the initial layers and more complex features in deeper layers.
- Convolutional layers are parameter-efficient, meaning fewer parameters can capture important spatial features.

How Does Convolution Work

- A filter is a small $n \times m$ matrix that we slide over the input image.
- For each position, we perform element-wise multiplication of the filter and the image region it covers, and then sum up the results.
- This sum is placed in a new matrix, creating the feature map.
- Filters are trained during the learning process, adapting to extract relevant features for the task at hand.

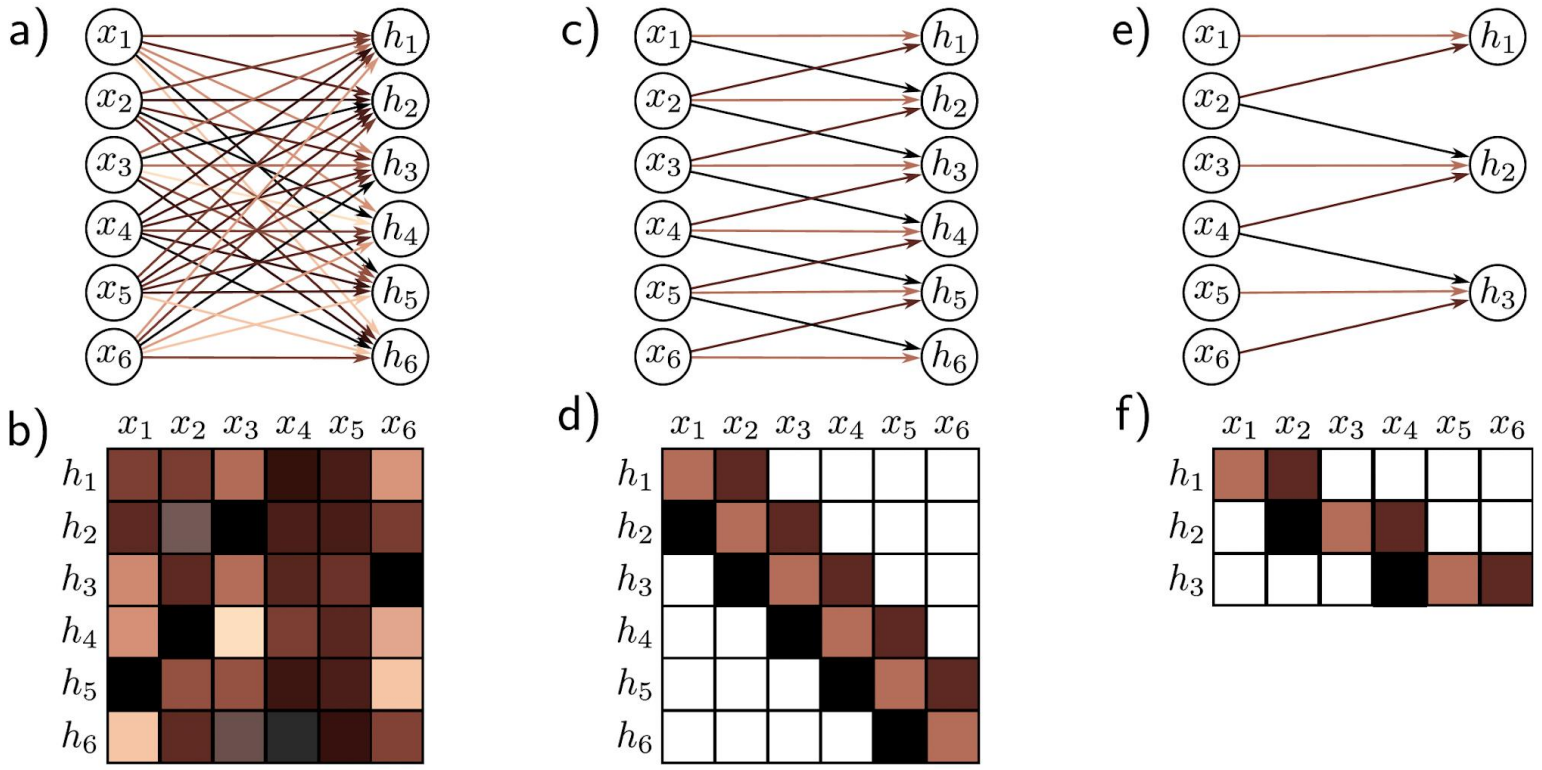
From Fully Connected to Convolutional Layers (1/3)

- Fully Connected Layers:
  - In a traditional neural network, each neuron in one layer is connected to every neuron in the subsequent layer.
  - While this works well for structured data, it results in a large number of parameters when dealing with high-dimensional data like images.
- Why Not Fully Connected for Images?
  - This architecture is not efficient for image data because it fails to take into account the spatial hierarchies between pixels.
  - The high number of parameters makes the network prone to overfitting.

From Fully Connected to Convolutional Layers (2/3)

- Introduction to Convolutional Layers:
  - Convolutional layers, on the other hand, maintain the spatial relationships between pixels by learning image features using small squares of input data.
  - These layers significantly reduce the number of parameters in the model, making it computationally more efficient.
- Efficiency Matters:
  - The reduced number of parameters not only makes the network computationally efficient but also helps in reducing overfitting.
  - This makes convolutional layers ideal for tasks involving image recognition and classification.

From Fully Connected to Convolutional Layers (3/3)



a) A fully connected layer has a weight connecting each input x to each hidden unit h (colored arrows) and a bias for each hidden unit (not shown).

b) Hence, the associated weight matrix Ω contains 36 weights relating the six inputs to the six hidden units.

c) A convolutional layer with kernel size three computes each hidden unit as the same weighted sum of the three neighboring inputs (arrows) plus a bias (not shown).

d) The weight matrix is a special case of the fully connected matrix where many weights are zero and others are repeated (same colors indicate same value, white indicates zero weight).

e) A convolutional layer with kernel size three and stride two computes a weighted sum at every other position.

f) This is also a special case of a fully connected network with a different sparse weight structure.

Source: Figure 10.4 in Understanding Deep Learning by Simon J.D. Prince.

Local Connectivity (1/3)

- Local Receptive Field:
  - In a convolutional layer, each neuron is connected only to a small region of the input volume.
  - This region is known as the neuron's "local receptive field," and it's where the neuron 'focuses' its attention.

Local Connectivity (2/3)

- Hierarchical Feature Learning:
  - The small receptive field allows the network to first learn low-level features like edges and corners.
  - As we go deeper into the network, these low-level features are combined to learn higher-level features like shapes or objects.

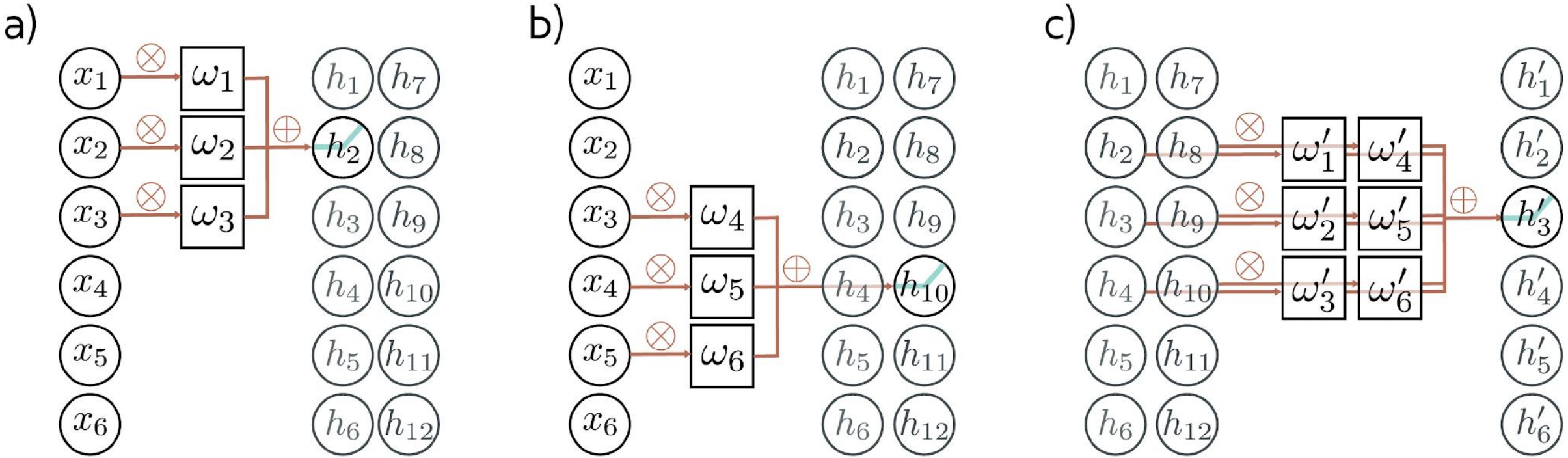
Local Connectivity (3/3)

- Parameter Sharing:
  - One of the key aspects of convolutional layers is that the same set of weights (also known as a filter or kernel) is used for different parts of the image, allowing the network to detect a particular feature regardless of where it appears in the input.

Channels (1/2)

- Channels are akin to different 'aspects' or 'dimensions' of the data.
- In an RGB image, each color (Red, Green, Blue) constitutes a channel.
- Clarify that in convolutional layers, each filter also has a depth that matches the number of input channels, enabling multi-dimensional feature extraction.

Channels (2/2)



Channels. Typically, multiple convolutions are applied to the input x and stored in channels.

a) A convolution is applied to create hidden units h1 to h6, which form the first channel.

b) A second convolution operation is applied to create hidden units h7 to h12, which form the second channel. The channels are stored in a 2D array H1 that contains all the hidden units in the first hidden layer.

c) If we add a further convolutional layer, there are now two channels at each input position. Here, the 1D convolution defines a weighted sum over both input channels at the three closest positions to create each new output channel.

Source: Figure 10.5 in Understanding Deep Learning by Simon J.D. Prince.

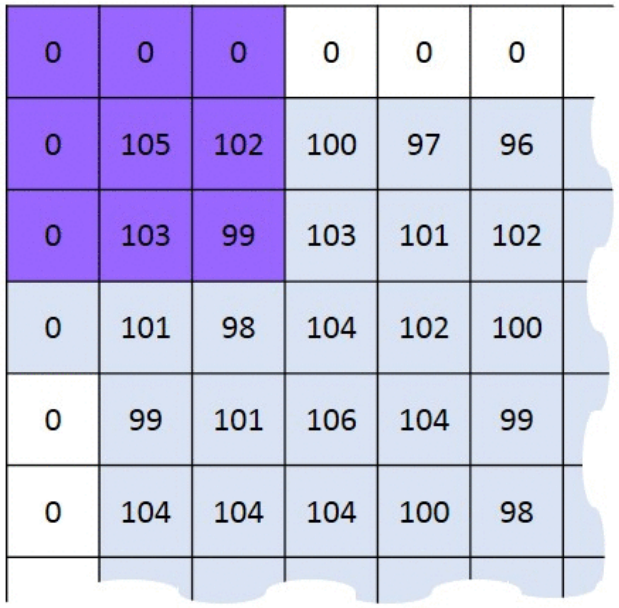
Filters and Kernels

- Introduce filters as the 'feature detectors' in CNNs.
- Explain that the term "kernel size" refers to the dimensions (width x height) of the filter.
- Larger kernels capture more global features, while smaller kernels capture more local features.

Convolution Operation

- The convolution operation involves taking a filter and sliding it over the input volume to produce a feature map.
- The filter's kernel size, as well as its depth (number of channels), define its shape.
- This feature map is a condensed form of the input, highlighting features like edges or textures depending on the filter used.

Convolution with 1 Channel of Input



0	0	0	0	0	0
0	105	102	100	97	96
0	103	99	103	101	102
0	101	98	104	102	100
0	99	101	106	104	99
0	104	104	104	100	98

Image Matrix

Kernel Matrix

0	-1	0
-1	5	-1
0	-1	0

$$0*0+0*-1+0*0$$
$$+0*-1+105*5+102*-1$$
$$+0*0+103*-1+99*0 = 320$$

320					

Output Matrix

Convolution with horizontal and vertical strides = 1

Source: https://bghojogh.github.io/assets/files/engg-6600-01-s23/DL_CNN.pdf

Stride

- The stride is the step size with which the filter moves across the input image.
- A stride of 1 moves the filter one pixel at a time, whereas a stride of 2 moves it two pixels at a time.
- A larger stride results in a smaller feature map.

Padding

- Padding involves adding extra pixels around the border of the input image. This is often done to control the size of the output feature maps, especially when you want the output volume to have the same spatial dimensions as the input.
- Types of Padding:
  - "Valid" padding means no padding is added, resulting in a smaller feature map.
  - "Same" padding means zeros are added such that the output feature map has the same dimensions as the input.

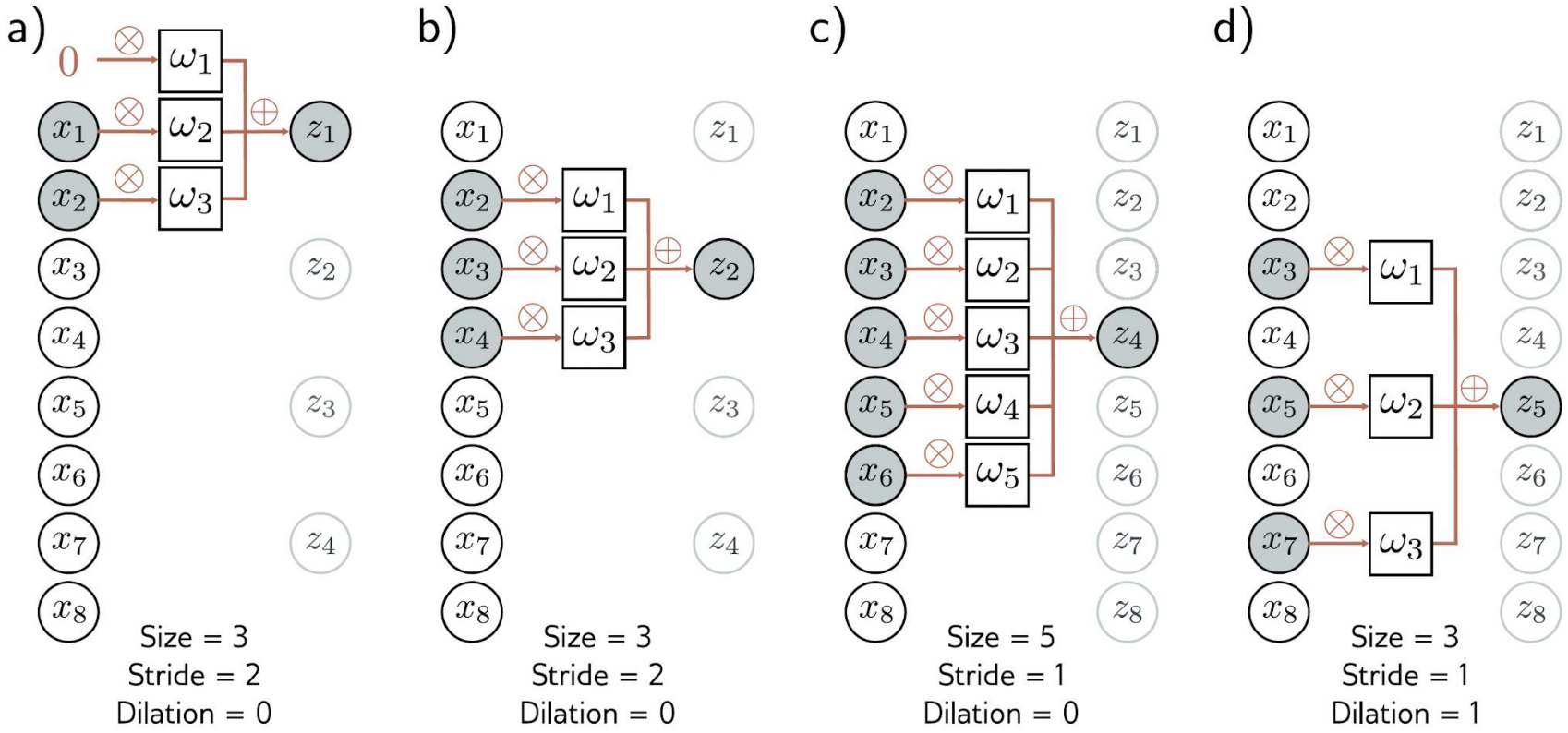
Dilation (1/2)

- Dilation is a technique used to expand the receptive field of filters in convolutional layers.
- It does so by introducing gaps between the pixels in the filter, effectively enlarging the filter's span across the input image without increasing the number of parameters.
- Dilation is useful for capturing more contextual information from the input image.
- This is particularly beneficial in tasks like image segmentation and object detection, where understanding the broader context is important.

Dilation (2/2)

- In dilated convolutions, the filter is applied to the input image with gaps.
  - For example, in a 3x3 filter with a dilation rate of 2, the filter would have the same field of view as a 5x5 filter but with only 9 parameters instead of 25.
- Dilated convolutions are commonly used in advanced CNN architectures like DeepLab for semantic image segmentation and WaveNet for audio generation.

Stride, Kernel Size, and Dilation



a) With a stride of two, we evaluate the kernel at every other position, so the first output z_1 is computed from a weighted sum centered at x_1, and b) the second output z_2 is computed from a weighted sum centered at x_3 and so on. c) The kernel size can also be changed. With a kernel size of five, we take a weighted sum of the nearest five inputs. d) In dilated or atrous convolution, we intersperse zeros in the weight vector to allow us to combine information over a large area using fewer weights.

Source: Figure 10.3 in Understanding Deep Learning by Simon J.D. Prince.

Pooling Layers

- Pooling layers typically come after convolutional layers in a CNN architecture.
- Their primary function is to reduce the spatial dimensions (width and height) of the input volume, thus lowering the computational cost for the network.
- By reducing the spatial dimensions, pooling layers help the CNN focus on the most critical features of the input.

Why Use Pooling

- Pooling layers introduce translational invariance to the network, enabling it to recognize features regardless of their location within the image.
- They help in preventing overfitting by reducing the spatial size of the convolved feature, effectively shortening the number of parameters and computations in the network.
- Pooling layers make the detection of features invariant to scale and orientation changes, providing a form of robustness.

Types of Pooling

- Max Pooling is the most commonly used pooling technique. It selects the maximum value from each cluster of the feature map to represent the region.
- Average Pooling computes the average value for each patch on the feature map.
- Global Average Pooling computes the average of all values in a feature map, resulting in a single scalar value per feature map.
- Min Pooling takes the minimum value from each cluster on the feature map, although it's less commonly used.

Examples for Types of Pooling

- Given a 2x2 region of a feature map as [1, 2; 3, 4]
  - Max pooling will select 4 as it's the largest value.
  - Average pooling will compute (1+2+3+4)/4 = 2.5
  - Global average would be (1+2+3+4)/4 = 2.5
  - Min pooling will select 1 as it's the smallest value

Overview of CNN Architecture

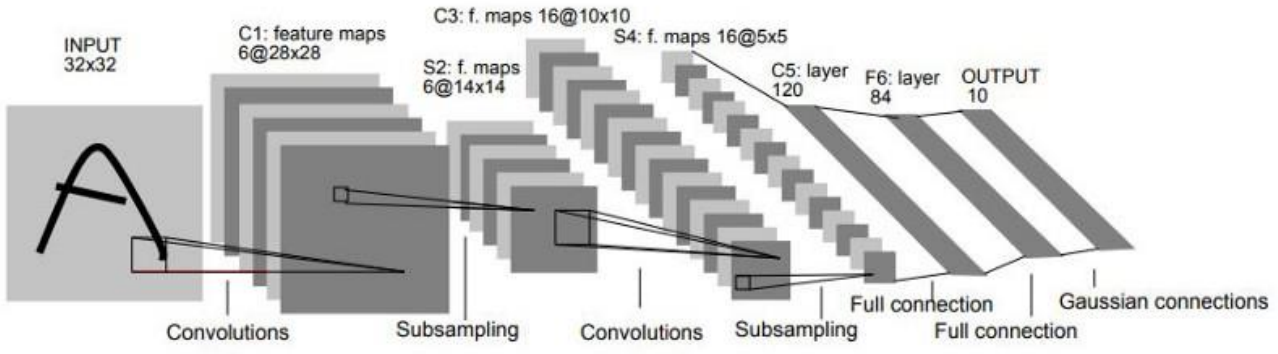
- CNNs are made up of a stack of layers designed to automatically and adaptively learn spatial hierarchies of features from input images.
- The architecture typically includes Input Layer, Convolutional Layers, Activation Layers, Pooling Layers, Fully Connected Layers, and Output Layer.
- The architecture defines the forward pass for making predictions and the backward pass for learning from errors.

Data Flow in a CNN

- During training, the input image is passed through the network layer-by-layer in a forward pass to produce a prediction.
- The prediction is then compared to the true label, and the error is computed.
- The error is backpropagated through the network in a backward pass, and the weights are updated.
- During inference (or testing), only the forward pass is executed to produce a prediction from the input image.

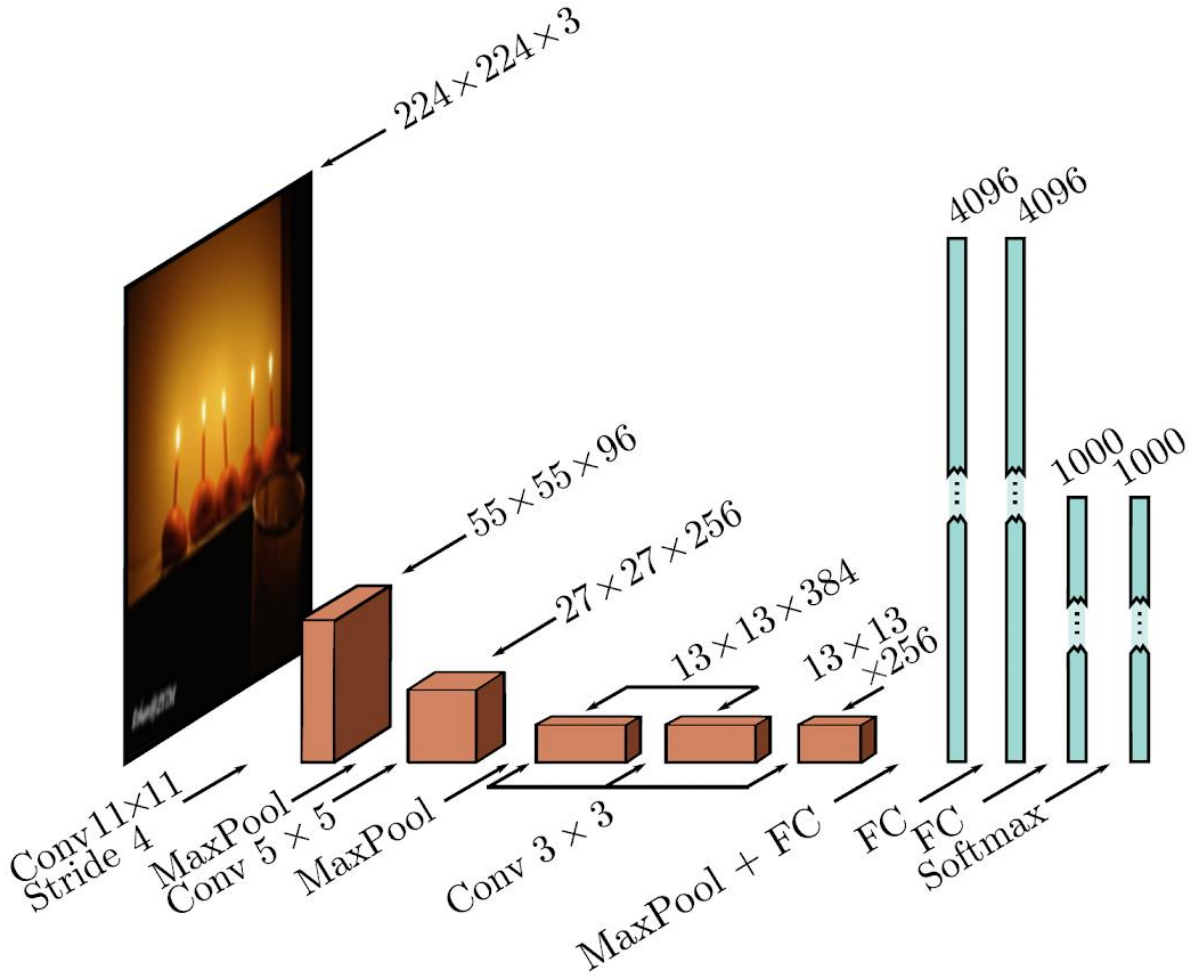
LeNet-5 Architecture

- Developed by Yann LeCun in 1998.
- Primarily used for digit recognition tasks.
- Consists of two sets of convolutional and average pooling layers, followed by a fully connected layer.



Source: https://www.datasciencecentral.com/lenet-5-a-classic-cnn-architecture/

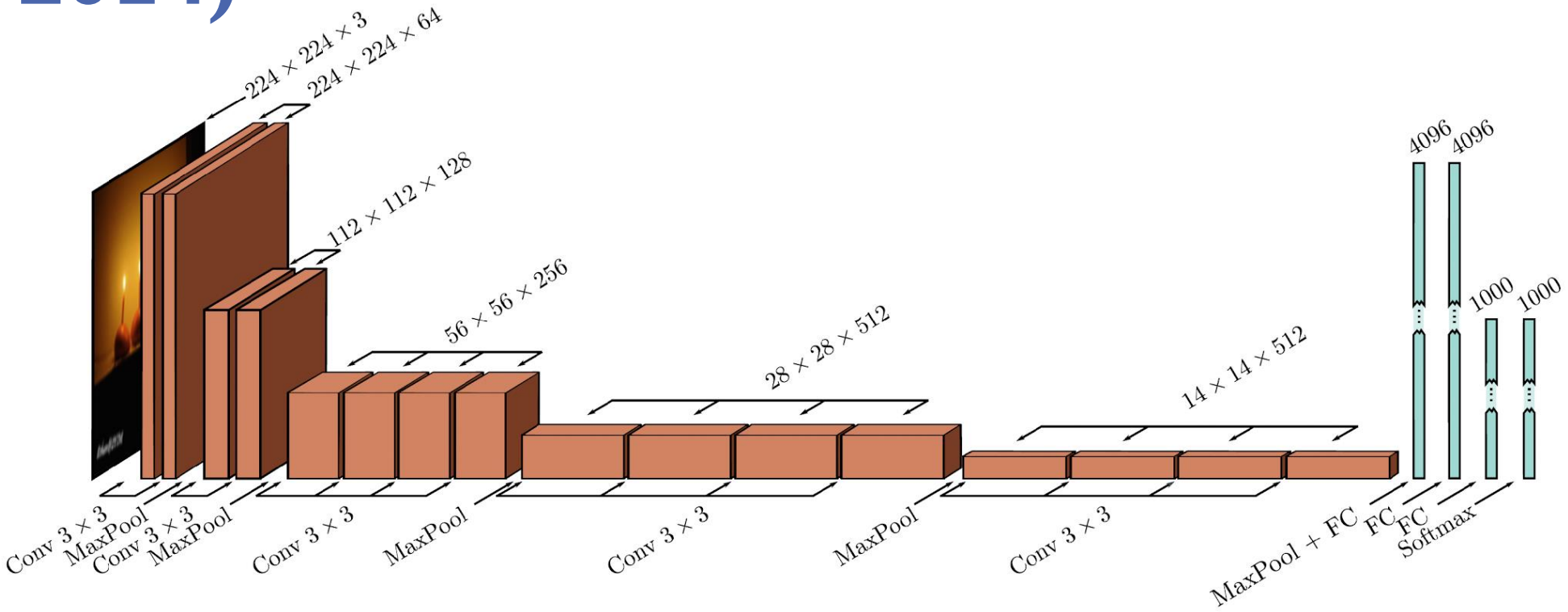
AlexNet (Krizhevsky et al., 2012)



The network maps a 224×224 color image to a 1000-dimensional vector representing class probabilities. The network first convolves with 11×11 kernels and stride 4 to create 96 channels. It decreases the resolution again using a max pool operation and applies a 5×5 convolutional layer. Another max pooling layer follows, and three 3×3 convolutional layers are applied. After a final max pooling operation, the result is vectorized and passed through three fully connected (FC) layers and finally the softmax layer.

Source: Figure 10.16 in Understanding Deep Learning by Simon J.D. Prince.

VGG Network (Simonyan & Zisserman, 2014)



This network consists of a series of convolutional layers and max pooling operations, in which the spatial scale of the representation gradually decreases, but the number of channels gradually increases. The hidden layer after the last convolutional operation is resized to a 1D vector and three fully connected layers follow. The network outputs 1000 activations corresponding to the class labels that are passed through a softmax function to create class probabilities.

Source: Figure 10.17 in Understanding Deep Learning by Simon J.D. Prince.

Hand-on Convolutional Neural Networks

```python
from tensorflow.keras.datasets import cifar10
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense

# Load data
(x_train, y_train), (x_test, y_test) = cifar10.load_data()

# Normalize data
x_train, x_test = x_train / 255.0, x_test / 255.0

# Build the model
model = Sequential([
    Conv2D(32, (3,3), activation='relu', input_shape=(32, 32, 3)),
    MaxPooling2D(2, 2),
    Conv2D(64, (3,3), activation='relu'),
    MaxPooling2D(2,2),
    Flatten(),
    Dense(128, activation='relu'),
    Dense(10, activation='softmax')
])

# Compile the model
model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy'])

# Train the model
history = model.fit(x_train, y_train, epochs=10, batch_size=64, validation_data=(x_test, y_test))

# Evaluate the model
test_loss, test_acc = model.evaluate(x_test, y_test)
print("Test accuracy: ", test_acc)
```

Thank you