Convolutional Neural Network Lecture Notes

Prepared By: Chandan Chaudhari
GitHub: chandanc5525

Contents

1 Introduction to Convolutional Neural Networks

1.1 What are Convolutional Neural Networks?

Convolutional Neural Networks (CNNs) are a class of deep neural networks specifically designed for processing structured grid data such as images. Unlike traditional neural networks, CNNs leverage spatial relationships between pixels through specialized operations that preserve spatial hierarchy while significantly reducing parameter count.

1.2 Core Architectural Principles

CNNs are built upon three fundamental principles:

Sparse Connectivity: Each neuron in a convolutional layer connects only to a small local region of the input volume, unlike dense connections in traditional networks. This local connectivity pattern captures local spatial correlations.

Parameter Sharing: The same filter weights are applied across all spatial positions of the input. This translation equivariance allows feature detection regardless of position while dramatically reducing parameters.

Hierarchical Feature Learning: CNNs learn features through multiple layers of abstraction:

- Early layers detect basic features (edges, corners)
- Middle layers combine these into patterns and shapes
- Deep layers recognize complex objects and structures

1.3 Biological Inspiration

CNNs draw inspiration from the visual cortex organization discovered by Hubel and Wiesel. The hierarchical processing in biological vision systems mirrors the layered architecture of CNNs, where simple cells respond to basic features and complex cells combine these responses.

2 Mathematical Intuition for CNNs

2.1 Convolution Operation Mathematics

The discrete 2D convolution operation between input $\mathbf{I}$ and kernel $\mathbf{K}$ is defined as:

$$(\mathbf{I} * \mathbf{K})(i,j) = \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} \mathbf{I}(i+m, j+n) \cdot \mathbf{K}(m,n) \tag{1}$$

For multiple input channels, the operation extends to:

$$\mathbf{Z}_k(i,j) = \sum_{c=1}^{C} \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} \mathbf{I}_c(i+m, j+n) \cdot \mathbf{K}_{k,c}(m,n) + b_k \tag{2}$$

2.2 Mathematical Properties

Translation Equivariance:

$$\text{If } \mathcal{T} \text{ is translation, then } \text{Conv}(\mathcal{T}(\mathbf{I}), \mathbf{K}) = \mathcal{T}(\text{Conv}(\mathbf{I}, \mathbf{K})) \tag{3}$$

Parameter Sharing: For a filter of size $F \times F$ with C input channels and K output channels:

$$\text{Parameters} = (F \times F \times C + 1) \times K \tag{4}$$

Compared to fully connected layer with input size $H \times W \times C$:

$$\text{FC Parameters} = (H \times W \times C) \times K \tag{5}$$

3 Applications of Convolutional Neural Networks

3.1 Computer Vision Applications

- **Image Classification**: Categorizing images into predefined classes
- **Object Detection**: Locating and classifying multiple objects within images
- **Semantic Segmentation**: Pixel-wise classification for scene understanding
- **Instance Segmentation**: Differentiating between instances of same class
- **Face Recognition**: Identity verification and recognition systems

3.2 Medical Imaging

- **Medical Diagnosis**: Detecting diseases from medical scans
- **Tumor Detection**: Identifying and segmenting tumors in MRI/CT scans
- **Cell Classification**: Analyzing microscopic images for pathology

3.3 Other Domains

- **Autonomous Driving**: Scene understanding and object detection
- **Document Analysis**: Handwriting recognition and document classification
- **Video Analysis**: Action recognition and video classification

4 Comparison of ANN vs CNN

Aspect	**Artificial Neural Networks (ANN)**	**Convolutional Neural Networks (CNN)**
Input Structure	Flattened 1D vectors	Structured 2D/3D grids (images)
Parameter Count	High: Each neuron connects to all inputs	Low: Local connectivity and parameter sharing
Spatial Information	Lost during flattening	Preserved through convolutional operations
Translation Invariance	No inherent invariance	Built-in through weight sharing
Feature Learning	Global feature combinations	Hierarchical local-to-global features
Computational Cost	High for image data	Optimized for spatial data
Use Cases	Tabular data, simple patterns	Images, videos, spatial data

Table 1: Comparison between ANN and CNN Architectures

5 Various Filters in CNN

5.1 Edge Detection Filters

5.1.1 Sobel Filter

Horizontal and vertical edge detection:

$$\mathbf{K}_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}, \quad \mathbf{K}_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} \tag{6}$$

Gradient magnitude:

$$G = \sqrt{(\mathbf{I} * \mathbf{K}_x)^2 + (\mathbf{I} * \mathbf{K}_y)^2} \tag{7}$$

5.1.2 Laplacian Filter

Second derivative for edge detection:

$$\mathbf{K} = \begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix} \tag{8}$$

5.2 Smoothing Filters

5.2.1 Gaussian Filter

Continuous form:

$$G_\sigma(x, y) = \frac{1}{2\pi\sigma^2} \exp\left(-\frac{x^2 + y^2}{2\sigma^2}\right) \tag{9}$$

Discrete approximation:

$$\mathbf{K} = \frac{1}{16}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix} \tag{10}$$

5.2.2 Mean Filter

Simple averaging:

$$\mathbf{K} = \frac{1}{9}\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix} \tag{11}$$

5.3 Sharpening Filters

5.3.1 Unsharp Masking

$$\mathbf{I}_{\text{sharp}} = \mathbf{I} + \lambda(\mathbf{I} - \mathbf{I}_{\text{blurred}}) \tag{12}$$

Direct implementation:

$$\mathbf{K} = \begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix} \tag{13}$$

6 In-depth Theory: Padding, Strides and Pooling

6.1 Padding: Comprehensive Analysis

Padding is a crucial operation in CNNs that controls the spatial dimensions of feature maps and preserves information at the borders of input images.

6.1.1 Mathematical Foundation of Padding

Given an input tensor $\mathbf{I} \in \mathbb{R}^{H \times W \times C}$ and padding P, the padded input $\mathbf{I}_{\text{pad}}$ is defined as:

$$\mathbf{I}_{\text{pad}}(i, j, c) = \begin{cases} \mathbf{I}(i - P, j - P, c) & \text{if } P \leq i < H + P \text{ and } P \leq j < W + P \\ 0 & \text{otherwise (zero padding)} \end{cases} \tag{14}$$

6.1.2 Types of Padding

Valid Padding (No Padding):

- No extra pixels added around the input
- Output dimensions decrease significantly
- Mathematical formulation:

$$H_{out} = H_{in} - F_h + 1 \tag{15}$$
$$W_{out} = W_{in} - F_w + 1 \tag{16}$$

- Information loss at borders
- Computational efficiency but reduced feature preservation

Same Padding:

- Padding calculated to maintain input dimensions
- Padding size: $P = \lfloor \frac{F-1}{2} \rfloor$
- Output dimensions equal input dimensions
- Preserves spatial information
- Most commonly used in practice

Full Padding:

- Maximum padding: $P = F - 1$
- Output dimensions: $H_{out} = H_{in} + F_h - 1$
- Used in transposed convolutions for upsampling
- Preserves all possible information but computationally expensive

6.1.3 Padding Implementation Example

```python
import tensorflow as tf
import numpy as np

def demonstrate_padding():
    input_tensor = tf.constant([[[[1], [2], [3]],
                                  [[4], [5], [6]],
                                  [[7], [8], [9]]]], dtype=tf.float32)

    print("Original input shape:", input_tensor.shape)
    print("Original input:\n", input_tensor.numpy().squeeze())

    valid_conv = tf.keras.layers.Conv2D(1, 3, padding='valid')
    same_conv = tf.keras.layers.Conv2D(1, 3, padding='same')

    valid_output = valid_conv(input_tensor)
    same_output = same_conv(input_tensor)

    print("\nValid padding output shape:", valid_output.shape)
    print("Same padding output shape:", same_output.shape)

    manual_padding = tf.pad(input_tensor, [[0,0], [1,1], [1,1], [0,0]])
    print("\nManually padded shape:", manual_padding.shape)

demonstrate_padding()
```

6.2 Strides: Comprehensive Analysis

Strides control the step size of the convolutional filter as it slides across the input, directly affecting the output dimensions and computational requirements.

6.2.1 Mathematical Formulation

For a convolution operation with stride S, the output at position (i, j) is computed as:

$$\mathbf{Z}(i,j,k) = \sum_{c=1}^{C} \sum_{m=0}^{F_h-1} \sum_{n=0}^{F_w-1} \mathbf{I}(i \cdot S + m, j \cdot S + n, c) \cdot \mathbf{K}(m,n,c,k) \tag{17}$$

6.2.2 Output Dimension Calculation

The general formula for output dimensions with stride S and padding P:

$$H_{out} = \left\lfloor \frac{H_{in} + 2P - F_h}{S} \right\rfloor + 1 \tag{18}$$

$$W_{out} = \left\lfloor \frac{W_{in} + 2P - F_w}{S} \right\rfloor + 1 \tag{19}$$

6.2.3 Stride Effects and Applications

Stride = 1:

- Dense sampling, maximum information preservation
- High computational cost
- Used in early layers for detailed feature extraction

Stride ¿ 1:

- Downsampling effect, reduces spatial dimensions
- Computational efficiency
- Increases receptive field rapidly
- Used for dimensionality reduction instead of pooling

6.2.4 Stride Implementation Example

```python
def demonstrate_strides():
    input_tensor = tf.random.normal([1, 8, 8, 3])

    print("Input shape:", input_tensor.shape)

    conv_stride_1 = tf.keras.layers.Conv2D(32, 3, strides=1, padding='same')
    conv_stride_2 = tf.keras.layers.Conv2D(32, 3, strides=2, padding='same')
    conv_stride_3 = tf.keras.layers.Conv2D(32, 3, strides=3, padding='same')

    output_1 = conv_stride_1(input_tensor)
    output_2 = conv_stride_2(input_tensor)
    output_3 = conv_stride_3(input_tensor)

    print("Stride 1 output shape:", output_1.shape)
    print("Stride 2 output shape:", output_2.shape)
    print("Stride 3 output shape:", output_3.shape)

    receptive_field_1 = 3
    receptive_field_2 = 3 + (3-1)*1
    receptive_field_3 = 3 + (3-1)*2

    print(f"Receptive field stride 1: {receptive_field_1}")
    print(f"Receptive field stride 2: {receptive_field_2}")
    print(f"Receptive field stride 3: {receptive_field_3}")

demonstrate_strides()
```

6.3 Pooling: Comprehensive Analysis

Pooling operations reduce spatial dimensions while preserving important features, providing translation invariance and reducing computational complexity.

6.3.1 Max Pooling Mathematical Foundation

For a pooling region $\mathcal{R}_{ij}$ of size $P_h \times P_w$:

$$\mathbf{P}(i,j,k) = \max_{(m,n)\in\mathcal{R}_{ij}} \mathbf{Z}(m,n,k) \tag{20}$$

Where $\mathcal{R}_{ij} = \{(m,n) : i \cdot S \leq m < i \cdot S + P_h, j \cdot S \leq n < j \cdot S + P_w\}$

6.3.2 Average Pooling Mathematical Foundation

$$\mathbf{P}(i,j,k) = \frac{1}{|\mathcal{R}_{ij}|} \sum_{(m,n)\in\mathcal{R}_{ij}} \mathbf{Z}(m,n,k) \tag{21}$$

6.3.3 Backpropagation Through Pooling

Max Pooling Backward Pass:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{Z}(m,n,k)} = \begin{cases} \frac{\partial \mathcal{L}}{\partial \mathbf{P}(i,j,k)} & \text{if } (m,n) = \arg\max_{(p,q)\in\mathcal{R}_{ij}} \mathbf{Z}(p,q,k) \\ 0 & \text{otherwise} \end{cases} \tag{22}$$

Average Pooling Backward Pass:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{Z}(m,n,k)} = \frac{1}{|\mathcal{R}_{ij}|} \frac{\partial \mathcal{L}}{\partial \mathbf{P}(i,j,k)} \quad \forall (m,n) \in \mathcal{R}_{ij} \tag{23}$$

6.3.4 Pooling Implementation Example

```python
def demonstrate_pooling():
    input_tensor = tf.constant([[[[1., 2., 3.],
                                  [4., 5., 6.],
                                  [7., 8., 9.]],
                                 [[9., 8., 7.],
                                  [6., 5., 4.],
                                  [3., 2., 1.]]]])

    print("Input shape:", input_tensor.shape)
    print("Input:\n", input_tensor.numpy().squeeze())

    max_pool = tf.keras.layers.MaxPooling2D(2, 2)
    avg_pool = tf.keras.layers.AveragePooling2D(2, 2)
    global_avg_pool = tf.keras.layers.GlobalAveragePooling2D()

    max_output = max_pool(input_tensor)
    avg_output = avg_pool(input_tensor)
    global_output = global_avg_pool(input_tensor)

    print("\nMax pooling output shape:", max_output.shape)
    print("Max pooling output:\n", max_output.numpy().squeeze())

    print("\nAverage pooling output shape:", avg_output.shape)
    print("Average pooling output:\n", avg_output.numpy().squeeze())

    print("\nGlobal average pooling output shape:", global_output.shape)
    print("Global average pooling output:", global_output.numpy())

demonstrate_pooling()
```

7 Pretrained Models in CNN

7.1 LeNet-5

- **Architecture**: 7 layers (2 convolutional, 2 pooling, 3 fully connected)
- **Input**: 32×32 grayscale images
- **Activation**: Tanh/Sigmoid
- **Application**: Handwritten digit recognition
- **Significance**: First successful CNN application

Mathematical structure:

$$\begin{aligned}\text{Input} &\rightarrow \text{Conv6} \rightarrow \text{AvgPool} \rightarrow \text{Conv16} \rightarrow \text{AvgPool} \quad (24)\\ &\rightarrow \text{FC120} \rightarrow \text{FC84} \rightarrow \text{Output10} \quad (25)\end{aligned}$$

7.2 AlexNet

- **Architecture**: 8 layers (5 convolutional, 3 fully connected)
- **Input**: 227×227×3 RGB images
- **Innovations**: ReLU, Dropout, Local Response Normalization
- **Parameters**: 60 million
- **Significance**: ImageNet 2012 winner, deep learning revolution

Key mathematical contributions:

$$\text{ReLU}(x) = \max(0, x) \quad \text{(addressed vanishing gradient)} \tag{26}$$

7.3 VGGNet

- **Architecture**: Uniform 3×3 convolutions with increasing depth
- **Variants**: VGG-16 (16 layers), VGG-19 (19 layers)
- **Parameters**: 138 million (VGG-16)
- **Innovation**: Depth importance demonstrated

Architecture pattern:

$$\text{Block}_l : [\text{Conv3×3}]_{N_l} \rightarrow \text{MaxPool2×2} \tag{27}$$

7.4 ResNet

- **Architecture**: Residual blocks with skip connections
- **Innovation**: Identity mapping solves vanishing gradient
- **Variants**: ResNet-18, ResNet-34, ResNet-50, ResNet-101, ResNet-152

Residual block mathematics:

$$\mathbf{y} = \mathcal{F}(\mathbf{x}, \{\mathbf{W}_i\}) + \mathbf{x} \tag{28}$$

Where $\mathcal{F}$ represents the residual mapping to learn.

7.5 Using Pretrained Models Implementation

```python
import tensorflow as tf
from tensorflow.keras.applications import VGG16, ResNet50, MobileNetV2
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Dense, GlobalAveragePooling2D
from tensorflow.keras.optimizers import Adam

def create_pretrained_model(model_name, input_shape, num_classes):
    if model_name == 'VGG16':
        base_model = VGG16(weights='imagenet', include_top=False, input_shape=
    input_shape)
    elif model_name == 'ResNet50':
        base_model = ResNet50(weights='imagenet', include_top=False, input_shape
    =input_shape)
    elif model_name == 'MobileNetV2':
        base_model = MobileNetV2(weights='imagenet', include_top=False,
    input_shape=input_shape)
    else:
        raise ValueError("Unsupported model name")

    base_model.trainable = False

    x = base_model.output
    x = GlobalAveragePooling2D()(x)
    x = Dense(1024, activation='relu')(x)
    predictions = Dense(num_classes, activation='softmax')(x)

    model = Model(inputs=base_model.input, outputs=predictions)

    return model

input_shape = (224, 224, 3)
num_classes = 10

vgg_model = create_pretrained_model('VGG16', input_shape, num_classes)
resnet_model = create_pretrained_model('ResNet50', input_shape, num_classes)
```

```

vgg_model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['
    accuracy'])

print("VGG16 Model Summary:")
vgg_model.summary()
```

8 Simple CNN Implementation

```
import tensorflow as tf
from tensorflow.keras import layers, models
import numpy as np

def create_simple_cnn(input_shape, num_classes):
    model = models.Sequential()

    model.add(layers.Conv2D(32, (3, 3), activation='relu', input_shape=
    input_shape))
    model.add(layers.MaxPooling2D((2, 2)))

    model.add(layers.Conv2D(64, (3, 3), activation='relu'))
    model.add(layers.MaxPooling2D((2, 2)))

    model.add(layers.Conv2D(64, (3, 3), activation='relu'))

    model.add(layers.Flatten())
    model.add(layers.Dense(64, activation='relu'))
    model.add(layers.Dense(num_classes, activation='softmax'))

    return model

input_shape = (28, 28, 1)
num_classes = 10
model = create_simple_cnn(input_shape, num_classes)

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

model.summary()
```

9 Project 1: Cat vs Dog Image Classification

```python
import tensorflow as tf
from tensorflow.keras.preprocessing.image import ImageDataGenerator
import os

base_dir = 'cats_and_dogs'
train_dir = os.path.join(base_dir, 'train')
validation_dir = os.path.join(base_dir, 'validation')

train_datagen = ImageDataGenerator(rescale=1./255)
test_datagen = ImageDataGenerator(rescale=1./255)

train_generator = train_datagen.flow_from_directory(
        train_dir,
        target_size=(150, 150),
        batch_size=20,
        class_mode='binary')

validation_generator = test_datagen.flow_from_directory(
        validation_dir,
        target_size=(150, 150),
        batch_size=20,
        class_mode='binary')

model = tf.keras.models.Sequential([
    tf.keras.layers.Conv2D(32, (3,3), activation='relu', input_shape=(150, 150,
    3)),
    tf.keras.layers.MaxPooling2D(2,2),
    tf.keras.layers.Conv2D(64, (3,3), activation='relu'),
    tf.keras.layers.MaxPooling2D(2,2),
    tf.keras.layers.Conv2D(128, (3,3), activation='relu'),
    tf.keras.layers.MaxPooling2D(2,2),
    tf.keras.layers.Conv2D(128, (3,3), activation='relu'),
    tf.keras.layers.MaxPooling2D(2,2),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(512, activation='relu'),
    tf.keras.layers.Dense(1, activation='sigmoid')
])

model.compile(loss='binary_crossentropy',
              optimizer=tf.keras.optimizers.RMSprop(learning_rate=1e-4),
              metrics=['accuracy'])

history = model.fit(
      train_generator,
      steps_per_epoch=100,
      epochs=30,
      validation_data=validation_generator,
      validation_steps=50)
```

10 Project 2: Fashion MNIST Dataset Analysis

```python
import tensorflow as tf
from tensorflow.keras import layers, models
import numpy as np
import matplotlib.pyplot as plt

fashion_mnist = tf.keras.datasets.fashion_mnist
(train_images, train_labels), (test_images, test_labels) = fashion_mnist.
    load_data()

class_names = ['T-shirt/top', 'Trouser', 'Pullover', 'Dress', 'Coat',
               'Sandal', 'Shirt', 'Sneaker', 'Bag', 'Ankle boot']

train_images = train_images.reshape((60000, 28, 28, 1))
test_images = test_images.reshape((10000, 28, 28, 1))

train_images = train_images.astype('float32') / 255
test_images = test_images.astype('float32') / 255

model = models.Sequential()
model.add(layers.Conv2D(32, (3, 3), activation='relu', input_shape=(28, 28, 1)))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))

model.add(layers.Flatten())
model.add(layers.Dense(64, activation='relu'))
model.add(layers.Dense(10, activation='softmax'))

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

history = model.fit(train_images, train_labels, epochs=10,
                    validation_data=(test_images, test_labels))

test_loss, test_acc = model.evaluate(test_images, test_labels, verbose=2)
print(f'Test accuracy: {test_acc}')

predictions = model.predict(test_images)

def plot_image(i, predictions_array, true_label, img):
    true_label, img = true_label[i], img[i]
    plt.grid(False)
    plt.xticks([])
    plt.yticks([])

    plt.imshow(img, cmap=plt.cm.binary)

    predicted_label = np.argmax(predictions_array)
    if predicted_label == true_label:
        color = 'blue'
    else:
        color = 'red'

    plt.xlabel("{} {:2.0f}% ({})".format(class_names[predicted_label],
                                100*np.max(predictions_array),
                                class_names[true_label]),
                                color=color)

def plot_value_array(i, predictions_array, true_label):
    true_label = true_label[i]
    plt.grid(False)
    plt.xticks(range(10))
    plt.yticks([])
    thisplot = plt.bar(range(10), predictions_array, color="#777777")
    plt.ylim([0, 1])
    predicted_label = np.argmax(predictions_array)

    thisplot[predicted_label].set_color('red')
    thisplot[true_label].set_color('blue')

num_rows = 5
num_cols = 3
num_images = num_rows * num_cols
plt.figure(figsize=(2*2*num_cols, 2*num_rows))
for i in range(num_images):
    plt.subplot(num_rows, 2*num_cols, 2*i+1)
    plot_image(i, predictions[i], test_labels, test_images)
    plt.subplot(num_rows, 2*num_cols, 2*i+2)
    plot_value_array(i, predictions[i], test_labels)
plt.tight_layout()
plt.show()
```