Fundamentals of CNNs and Hierarchical Feature Learning

Course:
Deep Learning with Tensorflow & Kersa 2

Developed by:
Mohammad Noorchenarboo

May 12, 2025

Current Section

1. Introduction: Why CNNs for Image Processing?
2. CNN Architecture: Components and Layer Types
3. Convolution Operation and Parameter Sharing
4. Pooling Layers and Spatial Reduction
5. Activation Functions in CNNs
6. Fully Connected Layers and Output Interpretation
7. Regularization in CNNs: Dropout, L2, BatchNorm
8. Hierarchical Feature Learning in CNNs
9. Comparing CNNs to ANNs
10. CNN Processing Pipeline
11. Real-World Architectures: LeNet, AlexNet, VGG, ResNet, Inception
12. CNN Architecture Implementations

Motivation: Why Do ANNs Fail on Image Data?

Traditional Artificial Neural Networks (ANNs) struggle with high-dimensional image data. Why?

- Lack of spatial awareness – every pixel treated independently.
- Large number of parameters – fully connected layers scale poorly.
- Poor translation invariance – small shifts can drastically change outputs.

High-Dimensional Input Problem

For a grayscale image of size 256×256, a single fully connected input layer needs $256 \times 256 = 65{,}536$ weights per neuron. This results in massive parameter growth for deeper networks.

Motivation: Why Do ANNs Fail on Image Data?

CNNs: A Better Approach

Convolutional Neural Networks (CNNs) leverage spatial structure via:

- Local connectivity
- Weight sharing (parameter efficiency)
- Hierarchical feature learning

Formula: Convolutional Output Size

Given an input of size $W \times H$, a kernel of size K, stride S, and padding P, the output size is:

$$O = \left\lfloor \frac{W - K + 2P}{S} \right\rfloor + 1$$

Motivation: Why Do ANNs Fail on Image Data?

Misconception Warning

CNNs are not invariant to all transformations by default. They exhibit some degree of translation invariance but not rotation or scale invariance without augmentation or architectural support.

Challenge	CNN Solution
Large number of parameters	Weight sharing reduces complexity
No spatial awareness	Local receptive fields capture spatial structure
No translation invariance	Pooling and local filters enable some invariance
Overfitting on small datasets	Regularization and fewer parameters help

Current Section

1. Introduction: Why CNNs for Image Processing?
2. CNN Architecture: Components and Layer Types
3. Convolution Operation and Parameter Sharing
4. Pooling Layers and Spatial Reduction
5. Activation Functions in CNNs
6. Fully Connected Layers and Output Interpretation
7. Regularization in CNNs: Dropout, L2, BatchNorm
8. Hierarchical Feature Learning in CNNs
9. Comparing CNNs to ANNs
10. CNN Processing Pipeline
11. Real-World Architectures: LeNet, AlexNet, VGG, ResNet, Inception
12. CNN Architecture Implementations

What Makes a CNN? Core Building Blocks

A Convolutional Neural Network (CNN) consists of layered components designed to process grid-like data (e.g., images). What are the key layers and their mathematical operations?

Core CNN Layers

- **Convolutional Layers** – extract local patterns using filters
- **Activation Functions** – introduce non-linearity
- **Pooling Layers** – downsample feature maps
- **Fully Connected Layers** – combine features into predictions
- **Normalization/Regularization** – improve generalization

What Makes a CNN? Core Building Blocks

Mathematical Operation: 2D Convolution

Let I be a 2D input image of size $W \times H$, and K a filter of size $f \times f$. The convolution at position (i, j) is:

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

This operation slides the kernel across the input to produce a feature map.

What Makes a CNN? Core Building Blocks

Simple 2D Convolution Example

```python
import numpy as np
from scipy.signal import convolve2d

I = np.array([[1, 2, 3],
              [4, 5, 6],
              [7, 8, 9]])

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

output = convolve2d(I, K, mode='valid')
print(output)
# Output:
# [[-4 -4]
#  [-4 -4]]
```

What Makes a CNN? Core Building Blocks

Output Interpretation

Each entry in the output corresponds to a local weighted sum using kernel K. The sign and magnitude capture edge or texture orientation.

Dimensional Explosion

Each convolution layer increases the number of feature maps, not just spatial dimensions. Unchecked, this leads to massive memory usage.

What Makes a CNN? Core Building Blocks

Padding vs No Padding

Without padding, output size shrinks:

$$O = \left\lfloor \frac{W - K}{S} + 1 \right\rfloor$$

With padding P, spatial size is preserved:

$$O = \left\lfloor \frac{W - K + 2P}{S} + 1 \right\rfloor$$

Best Practice: Use Powers of 2

Feature map sizes are often reduced by factors of 2 (e.g., $64\times64 \rightarrow 32\times32 \rightarrow 16\times16$) for regularity and memory efficiency.

What Makes a CNN? Core Building Blocks

Layer Type	Mathematical Description
Convolutional Layer	$y_{i,j} = \sum_{m,n} I_{i+m,j+n} \cdot K_{m,n}$
Activation Layer (ReLU)	$f(x) = \max(0, x)$
Pooling Layer	$y_{i,j} = \max_{(m,n)\in\Omega(i,j)} x_{m,n}$ or average
Fully Connected Layer	$\mathbf{y} = \mathbf{W}\mathbf{x} + \mathbf{b}$
Dropout Layer	Randomly sets a fraction p of inputs to 0 during training

Current Section

1 Introduction: Why CNNs for Image Processing?
2 CNN Architecture: Components and Layer Types
3 Convolution Operation and Parameter Sharing
4 Pooling Layers and Spatial Reduction
5 Activation Functions in CNNs
6 Fully Connected Layers and Output Interpretation
7 Regularization in CNNs: Dropout, L2, BatchNorm
8 Hierarchical Feature Learning in CNNs
9 Comparing CNNs to ANNs
10 CNN Processing Pipeline
11 Real-World Architectures: LeNet, AlexNet, VGG, ResNet, Inception
12 CNN Architecture Implementations

How Do CNNs Reduce Parameters?

The convolutional layer is central to CNNs. It applies the same set of weights (kernel) across the input – a technique known as **parameter sharing**.

Key Insight: Local Receptive Fields + Shared Weights

Each neuron in a convolutional layer is connected only to a small region (receptive field) of the input, and the same kernel is applied across the entire input.

How Do CNNs Reduce Parameters?

Mathematical Breakdown: Convolution with Shared Weights

Let input $X \in \mathbb{R}^{W \times H}$, kernel $K \in \mathbb{R}^{f \times f}$:

$$Y_{i,j} = \sum_{m=0}^{f-1} \sum_{n=0}^{f-1} X_{i+m,j+n} \cdot K_{m,n}$$

This is computed over all valid (i, j) positions – same weights K are reused!

Watch for Shape Mismatch

To convolve multiple channels (e.g., RGB), kernel K must have shape $f \times f \times C_{\text{in}}$, where C_{in} is the number of input channels.

How Do CNNs Reduce Parameters?

Parameter Reduction: Fully Connected vs CNN

```
# Fully connected layer: 100x100 input, 500 neurons
fc_params = 100 * 100 * 500  # = 5,000,000

# CNN: 5 filters of size 3x3, 1 input channel
cnn_params = 5 * 3 * 3 * 1   # = 45
```

CNN Advantage

CNNs drastically reduce learnable parameters by:

- Reusing filters across the spatial dimensions
- Connecting locally, not globally
- Using the same kernel weights for all input positions

How Do CNNs Reduce Parameters?

Numerical Example

Input: $X \in \mathbb{R}^{4\times 4}$

$$X = \begin{bmatrix} 1 & 2 & 3 & 0 \\ 4 & 5 & 6 & 1 \\ 7 & 8 & 9 & 0 \\ 1 & 2 & 3 & 4 \end{bmatrix}, \quad K = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$$

Convolution result at top-left position:

$$(1 \cdot 1 + 2 \cdot 0 + 4 \cdot 0 + 5 \cdot (-1)) = 1 - 5 = -4$$

How Do CNNs Reduce Parameters?

Loss of Global Context

Convolution layers only capture local patterns. Global relationships must be learned via stacking multiple layers or using larger receptive fields.

Concept	**Explanation / Formula**
Parameter Sharing	Same weights reused across positions: $K_{m,n}$ fixed for all (i, j)
Local Receptive Field	Each neuron sees only a region of input, e.g., 3×3 patch
Parameters (CNN)	$\#\text{filters} \times f \times f \times C_{\text{in}}$
Parameters (FC)	$W \cdot H \cdot N_{\text{neurons}}$

Current Section

1. Introduction: Why CNNs for Image Processing?
2. CNN Architecture: Components and Layer Types
3. Convolution Operation and Parameter Sharing
4. Pooling Layers and Spatial Reduction
5. Activation Functions in CNNs
6. Fully Connected Layers and Output Interpretation
7. Regularization in CNNs: Dropout, L2, BatchNorm
8. Hierarchical Feature Learning in CNNs
9. Comparing CNNs to ANNs
10. CNN Processing Pipeline
11. Real-World Architectures: LeNet, AlexNet, VGG, ResNet, Inception
12. CNN Architecture Implementations

Why Pooling? Simplifying Representations

Pooling layers are used to reduce the spatial size of feature maps while retaining important information.

Purpose of Pooling Layers

- Reduce dimensionality and computational cost
- Introduce translation invariance
- Help prevent overfitting

Why Pooling? Simplifying Representations

Max Pooling: Most Common Variant

Given a feature map X and pooling window size $p \times p$, max pooling computes:

$$Y_{i,j} = \max_{0 \leq m < p, 0 \leq n < p} X_{i \cdot p+m, j \cdot p+n}$$

Example: 2×2 max pooling reduces a 4×4 input to 2×2.

Why Pooling? Simplifying Representations

Numerical Example: Max Pooling

```python
import numpy as np
from tensorflow.keras.layers import MaxPooling2D
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Input

model = Sequential([
    Input(shape=(4, 4, 1)),
    MaxPooling2D(pool_size=(2, 2))
])

# Example input: batch of 1 image
X = np.array([[[[1], [2], [3], [4]],
               [[5], [6], [7], [8]],
               [[9], [10], [11], [12]],
               [[13], [14], [15], [16]]]])

output = model.predict(X)
print(output.squeeze())
# Output:
# [[ 6.  8.]
#  [14. 16.]]
```

Why Pooling? Simplifying Representations

Interpretation

Each 2×2 region is reduced to its maximum value. This emphasizes prominent features.

Drawback of Pooling

Pooling is a hand-crafted operation – it is non-learnable. Overuse can lead to information loss.

Why Pooling? Simplifying Representations

Average Pooling vs Max Pooling

Max pooling highlights the most active features, while average pooling smooths representations. Choice depends on task-specific needs.

Formula: Output Size After Pooling

For input size $W \times H$, pooling window size P, stride S:

$$O = \left\lfloor \frac{W - P}{S} + 1 \right\rfloor$$

With $W = 4$, $P = 2$, $S = 2$, we get $O = 2$.

Why Pooling? Simplifying Representations

Pooling Type	Effect
Max Pooling	Keeps strongest activation in region; emphasizes feature presence
Average Pooling	Computes average; smoother but less discriminative
Global Average Pooling	Reduces each feature map to a single number (used before FC layer)
Stride > 1	Introduces downsampling, even without overlap

Current Section

1. Introduction: Why CNNs for Image Processing?
2. CNN Architecture: Components and Layer Types
3. Convolution Operation and Parameter Sharing
4. Pooling Layers and Spatial Reduction
5. Activation Functions in CNNs
6. Fully Connected Layers and Output Interpretation
7. Regularization in CNNs: Dropout, L2, BatchNorm
8. Hierarchical Feature Learning in CNNs
9. Comparing CNNs to ANNs
10. CNN Processing Pipeline
11. Real-World Architectures: LeNet, AlexNet, VGG, ResNet, Inception
12. CNN Architecture Implementations

Why Activation Functions?

Without activation functions, CNNs behave as linear systems – no matter how deep. We need non-linearity to model complex functions.

Role of Activation Functions

- Introduce non-linearity
- Enable networks to learn complex mappings
- Applied element-wise after convolution or FC layers

Why Activation Functions?

Most Common: ReLU (Rectified Linear Unit)

$$f(x) = \max(0, x)$$

Properties:

- Efficient to compute
- Avoids vanishing gradient (for $x > 0$)

Why Activation Functions?

Visualizing ReLU vs Sigmoid

```
import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(-10, 10, 100)
relu = np.maximum(0, x)
sigmoid = 1 / (1 + np.exp(-x))

plt.plot(x, relu, label='ReLU')
plt.plot(x, sigmoid, label='Sigmoid')
plt.legend()
plt.grid()
plt.title("ReLU vs Sigmoid")
plt.show()
```

Output Interpretation

ReLU returns 0 for negative values and linear for positive ones. **Sigmoid** squashes input to $(0, 1)$ range.

Why Activation Functions?

ReLU Caveat: Dying Neurons

If too many activations fall below zero, gradients become zero – neurons stop learning. Consider using Leaky ReLU or ELU.

Other Activation Functions

- **Sigmoid**: $f(x) = \frac{1}{1+e^{-x}}$ – used in binary classification
- **Tanh**: $f(x) = \tanh(x)$ – centered around 0, but prone to vanishing gradients
- **Leaky ReLU**: $f(x) = \max(0.01x, x)$ – mitigates dying neuron problem
- **Softmax**: $\text{softmax}(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}}$ – used in multiclass classification

Why Activation Functions?

Vanishing Gradient Problem

Sigmoid and Tanh compress large inputs into small derivatives, leading to small gradient updates:

$$\frac{d}{dx}\sigma(x) = \sigma(x)(1-\sigma(x))$$

Activation	**Formula / Use Case**
ReLU	$f(x) = \max(0, x)$ – fast, common in hidden layers
Leaky ReLU	$f(x) = \max(0.01x, x)$ – avoids dying neurons
Sigmoid	$f(x) = \frac{1}{1+e^{-x}}$ – used in output layer for binary classification
Tanh	$f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$ – zero-centered
Softmax	$f(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}}$ – multiclass output probabilities

Current Section

1. Introduction: Why CNNs for Image Processing?
2. CNN Architecture: Components and Layer Types
3. Convolution Operation and Parameter Sharing
4. Pooling Layers and Spatial Reduction
5. Activation Functions in CNNs
6. Fully Connected Layers and Output Interpretation
7. Regularization in CNNs: Dropout, L2, BatchNorm
8. Hierarchical Feature Learning in CNNs
9. Comparing CNNs to ANNs
10. CNN Processing Pipeline
11. Real-World Architectures: LeNet, AlexNet, VGG, ResNet, Inception
12. CNN Architecture Implementations

What Happens After Convolutions and Pooling?

After the convolutional and pooling layers extract features, fully connected (FC) layers combine these into a final output – typically for classification or regression.

Purpose of Fully Connected Layers

- Map learned features to decision boundaries
- Flatten spatial dimensions into a 1D vector
- Perform high-level reasoning based on global features

What Happens After Convolutions and Pooling?

Mathematical Formulation

Let flattened input vector be $\mathbf{x} \in \mathbb{R}^n$, weight matrix $\mathbf{W} \in \mathbb{R}^{m \times n}$, bias $\mathbf{b} \in \mathbb{R}^m$:

$$\mathbf{y} = \mathbf{W}\mathbf{x} + \mathbf{b}$$

If softmax is applied:

$$\text{softmax}(y_i) = \frac{e^{y_i}}{\sum_j e^{y_j}} \quad \text{(multiclass output probabilities)}$$

What Happens After Convolutions and Pooling?

CNN Output Layer for Classification

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

model = Sequential([
    Conv2D(32, kernel_size=(3,3), activation='relu',
        input_shape=(28,28,1)),
    Flatten(),
    Dense(128, activation='relu'),
    Dense(10, activation='softmax')  # 10-class classification
])

model.summary()
# Output shows:
# Flatten layer reduces spatial dims (e.g., 26x26x32 -> 21632)
# Dense(128): FC layer with 128 neurons
# Dense(10): final class probabilities
```

What Happens After Convolutions and Pooling?

Interpretation

The Dense layer learns to combine all extracted features to predict class scores. Softmax ensures the output sums to 1, representing a valid probability distribution.

High Parameter Count

Fully connected layers have many weights and often dominate the model size:

$$\text{Parameters} = \text{Input Size} \times \text{Neurons} + \text{Biases}$$

E.g., input of size 4096 to 1024 neurons: $4096 \times 1024 + 1024 = 4,195,328$ parameters.

What Happens After Convolutions and Pooling?

Overfitting Risk in Dense Layers

Dense layers are highly expressive and prone to overfitting, especially if used after small datasets or insufficient regularization.

Output Layer Activation: Use Cases

- **Softmax**: Multiclass classification (one-hot labels)
- **Sigmoid**: Binary or multilabel classification
- **Linear**: Regression

What Happens After Convolutions and Pooling?

Component	Role in CNN
Flatten Layer	Converts 3D feature maps into 1D vector
Dense Layer	Applies matrix multiplication + bias: $\mathbf{Wx} + \mathbf{b}$
Softmax Output	Converts raw scores to class probabilities
Fully Connected Parameters	$n \cdot m + m$ where n = inputs, m = neurons
Risk	High parameter count leads to overfitting without regularization

Current Section

1. Introduction: Why CNNs for Image Processing?
2. CNN Architecture: Components and Layer Types
3. Convolution Operation and Parameter Sharing
4. Pooling Layers and Spatial Reduction
5. Activation Functions in CNNs
6. Fully Connected Layers and Output Interpretation
7. **Regularization in CNNs: Dropout, L2, BatchNorm**
8. Hierarchical Feature Learning in CNNs
9. Comparing CNNs to ANNs
10. CNN Processing Pipeline
11. Real-World Architectures: LeNet, AlexNet, VGG, ResNet, Inception
12. CNN Architecture Implementations

Controlling Overfitting in Deep CNNs

CNNs are powerful, but they can easily overfit – especially with small or noisy datasets. Regularization techniques help control this.

Why Regularization?

- Reduces model variance
- Improves generalization
- Controls the complexity of the learned function

Controlling Overfitting in Deep CNNs

L2 Regularization (Weight Decay)

Adds a penalty term to the loss function:

$$L_{\text{total}} = L_{\text{data}} + \lambda \sum_i w_i^2$$

Where λ controls the strength of the penalty. Encourages smaller weights.

L2 Regularization in Keras

```
from tensorflow.keras import regularizers
from tensorflow.keras.layers import Dense

Dense(128, activation='relu',
    kernel_regularizer=regularizers.l2(0.001))
```

Controlling Overfitting in Deep CNNs

Effect of L2

Penalizing large weights makes the model rely on broader, more stable patterns rather than memorization.

Dropout

Randomly "drops" a fraction of neurons during training:

$$\text{During training: } x_i' = \begin{cases} 0 & \text{with probability } p \\ \frac{x_i}{1-p} & \text{otherwise} \end{cases}$$

At test time, no dropout is applied – outputs are scaled automatically.

Controlling Overfitting in Deep CNNs

Using Dropout in Keras

```python
from tensorflow.keras.layers import Dropout

Dropout(0.5)  # Drop 50% of inputs during training
```

Caution: Dropout Placement

Dropout is typically used after Dense layers, not convolutional layers. Using it on feature maps may degrade spatial structure.

Controlling Overfitting in Deep CNNs

Batch Normalization

Normalizes activations per mini-batch:

$$\hat{x} = \frac{x - \mu}{\sqrt{\sigma^2 + \varepsilon}}, \quad y = \gamma \hat{x} + \beta$$

This improves gradient flow and speeds up convergence.

Batch Normalization in Keras

```python
from tensorflow.keras.layers import BatchNormalization

model.add(BatchNormalization())
```

Controlling Overfitting in Deep CNNs

BatchNorm Benefits

- Reduces internal covariate shift
- Acts as a regularizer
- Allows for higher learning rates

Potential Conflicts

Using Dropout and BatchNorm together can create optimization issues. Evaluate their effectiveness separately during tuning.

Regularization Method	Effect / Formula
L2 Regularization	$\lambda \sum w_i^2$ – penalizes large weights
Dropout	Randomly disables neurons: $x_i' = 0$ with prob. p
Batch Normalization	Normalizes and re-centers activations per batch
Weight Constraint	Enforces upper bound on layer weights

Current Section

1. Introduction: Why CNNs for Image Processing?
2. CNN Architecture: Components and Layer Types
3. Convolution Operation and Parameter Sharing
4. Pooling Layers and Spatial Reduction
5. Activation Functions in CNNs
6. Fully Connected Layers and Output Interpretation
7. Regularization in CNNs: Dropout, L2, BatchNorm
8. **Hierarchical Feature Learning in CNNs**
9. Comparing CNNs to ANNs
10. CNN Processing Pipeline
11. Real-World Architectures: LeNet, AlexNet, VGG, ResNet, Inception
12. CNN Architecture Implementations

How Do CNNs Understand Images?

CNNs learn features in a hierarchical fashion – from simple patterns to complex abstractions.

Definition: Hierarchical Feature Learning

CNNs progressively learn to detect features at increasing levels of abstraction:

- **Low-level:** Edges, corners, textures
- **Mid-level:** Patterns, shapes, motifs
- **High-level:** Object parts, semantic features

How Do CNNs Understand Images?

Layered Interpretation

Assume input image $X \in \mathbb{R}^{W \times H \times 3}$

$$
\begin{aligned}
\text{Layer 1 (Conv)} &\rightarrow \text{Edge detectors (filters like Sobel)} \\
\text{Layer 2 (Conv + Pool)} &\rightarrow \text{Textures, corners} \\
\text{Layer 3+ (Conv)} &\rightarrow \text{Motifs, object shapes} \\
\text{Final Dense Layer} &\rightarrow \text{Class predictions}
\end{aligned}
$$

How Do CNNs Understand Images?

Visualizing Hierarchical Features

```python
from tensorflow.keras.applications import VGG16
from tensorflow.keras.models import Model
import matplotlib.pyplot as plt
import numpy as np

model = VGG16(weights='imagenet', include_top=False)
layer_outputs = [layer.output for layer in model.layers[:8]]
feature_model = Model(inputs=model.input, outputs=layer_outputs)

img = np.random.rand(1, 224, 224, 3)
features = feature_model.predict(img)

for i, fmap in enumerate(features):
    print(f"Layer {i+1} shape: {fmap.shape}")
# Output: Progressively deeper layers, reduced spatial size,
    increased depth
```

How Do CNNs Understand Images?

Interpretation

Earlier layers learn generic visual filters. Deeper layers encode task-specific representations. This is why pretrained CNNs work well for transfer learning.

Loss of Fine Detail

While deeper layers capture semantics, spatial resolution is reduced via pooling – precise localization is lost unless architecture includes upsampling.

How Do CNNs Understand Images?

Beware: Shallow Networks Can't Learn Semantics

A CNN with few layers will only learn low-level patterns. Hierarchy is built through sufficient depth and non-linearity.

Numerical Analogy: Feature Maps

If a 32×32 input passes through:

- Conv layer with 3×3 filters $\rightarrow$ 32 feature maps
- Pooling (2x2) $\rightarrow$ size becomes 16×16
- Deeper conv + pool $\rightarrow$ size 8×8, more feature maps (64, 128, etc.)

The depth increases while width and height decrease – building depth-wise abstractions.

How Do CNNs Understand Images?

Feature Level	Examples
Low-Level Features	Lines, edges, gradients, blobs
Mid-Level Features	Corners, curves, textures, repeated patterns
High-Level Features	Eyes, noses, wheels, text, object parts
Output Layer	Semantic class predictions (dog, car, digit, etc.)

Current Section

1. Introduction: Why CNNs for Image Processing?
2. CNN Architecture: Components and Layer Types
3. Convolution Operation and Parameter Sharing
4. Pooling Layers and Spatial Reduction
5. Activation Functions in CNNs
6. Fully Connected Layers and Output Interpretation
7. Regularization in CNNs: Dropout, L2, BatchNorm
8. Hierarchical Feature Learning in CNNs
9. **Comparing CNNs to ANNs**
10. CNN Processing Pipeline
11. Real-World Architectures: LeNet, AlexNet, VGG, ResNet, Inception
12. CNN Architecture Implementations

Why Are CNNs Superior for Image Data?

Artificial Neural Networks (ANNs) and Convolutional Neural Networks (CNNs) differ primarily in how they handle spatial structure and parameter efficiency.

Core Difference

ANNs: Fully connected layers treat each input feature independently.
CNNs: Preserve spatial relationships using local filters and shared weights.

Why Are CNNs Superior for Image Data?

Parameter Comparison: ANN vs CNN

Case: 64×64 grayscale image (4096 inputs)
ANN:

1 hidden layer with 512 units $\rightarrow 4096 \times 512 + 512 = 2,097,664$ parameters

CNN:

- 32 filters, size 3×3
- Parameters = $32 \times 3 \times 3 = 288$

CNN uses 7,000x fewer parameters!

Why Are CNNs Superior for Image Data?

Empirical Comparison: ANN vs CNN on MNIST

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Conv2D, Flatten, Input
from tensorflow.keras.datasets import mnist
from tensorflow.keras.utils import to_categorical

(x_train, y_train), (_, _) = mnist.load_data()
x_train = x_train.reshape(-1, 28*28).astype('float32') / 255
y_train = to_categorical(y_train)

# ANN model
ann = Sequential([
    Input(shape=(784,)),
    Dense(512, activation='relu'),
    Dense(10, activation='softmax')
])
ann.compile(optimizer='adam', loss='categorical_crossentropy',
    metrics=['accuracy'])
ann.summary()
```

Why Are CNNs Superior for Image Data?

Output Interpretation

ANNs are simple but inefficient with high-dimensional data. CNNs are more compact, efficient, and accurate for structured inputs like images.

ANN Limitations for Image Data

- Flattening destroys spatial locality
- Large number of parameters lead to overfitting
- Poor generalization to translated inputs

Misconception: CNNs Always Better

For non-grid data (e.g., tabular), ANNs or decision trees may outperform CNNs. Choose architecture based on data modality.

Why Are CNNs Superior for Image Data?

CNN Advantages Recap

- Fewer parameters (weight sharing)
- Built-in spatial hierarchy
- Better generalization with less data
- Translation invariance via pooling

Attribute	**ANN** vs **CNN**
Parameter Count	ANN: Very high (FC layers); CNN: Very low (shared filters)
Spatial Awareness	ANN: None; CNN: Preserves structure
Feature Extraction	ANN: Manual or poor; CNN: Automatic and hierarchical
Use Case	ANN: Tabular, signal; CNN: Images, spatial data
Robustness to Shift	ANN: Low; CNN: Higher due to local patterns and pooling

Current Section

1. Introduction: Why CNNs for Image Processing?
2. CNN Architecture: Components and Layer Types
3. Convolution Operation and Parameter Sharing
4. Pooling Layers and Spatial Reduction
5. Activation Functions in CNNs
6. Fully Connected Layers and Output Interpretation
7. Regularization in CNNs: Dropout, L2, BatchNorm
8. Hierarchical Feature Learning in CNNs
9. Comparing CNNs to ANNs
10. **CNN Processing Pipeline**
11. Real-World Architectures: LeNet, AlexNet, VGG, ResNet, Inception
12. CNN Architecture Implementations

How Does Data Flow Through a CNN?

Let's break down the complete data flow and transformations in a CNN model.

End-to-End Pipeline Overview

1. Input Image $\rightarrow$ Preprocessing (resize, normalize)
2. Convolution Layers $\rightarrow$ Local feature extraction
3. Activation Functions $\rightarrow$ Non-linear transformations
4. Pooling Layers $\rightarrow$ Spatial dimension reduction
5. Flatten Layer $\rightarrow$ Convert 3D to 1D
6. Fully Connected Layers $\rightarrow$ Feature combination
7. Output Layer $\rightarrow$ Final prediction (softmax/sigmoid/linear)

How Does Data Flow Through a CNN?

Mathematical Flow Through the Network

$$\text{Input: } X \in \mathbb{R}^{W \times H \times C}$$

$$\text{Conv: } X_1 = \sigma(W_1 * X + b_1) \quad \text{(non-linearity)}$$

$$\text{Pool: } X_2 = \text{Pool}(X_1)$$

$$\text{Flatten: } x = \text{vec}(X_2) \quad \text{(vectorize)}$$

$$\text{Dense: } y = \text{softmax}(W_2 x + b_2)$$

How Does Data Flow Through a CNN?

Keras Implementation of Full CNN Pipeline

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

model = Sequential([
    Input(shape=(28, 28, 1)),
    Conv2D(32, (3, 3), activation='relu'),
    MaxPooling2D(pool_size=(2, 2)),
    Conv2D(64, (3, 3), activation='relu'),
    MaxPooling2D(pool_size=(2, 2)),
    Flatten(),
    Dense(128, activation='relu'),
    Dense(10, activation='softmax')
])

model.summary()
```

How Does Data Flow Through a CNN?

Typical CNN Layer Progression

- Input: $(28, 28, 1)$
- After Conv2D(32): $(26, 26, 32)$
- After MaxPooling2D: $(13, 13, 32)$
- After Conv2D(64): $(11, 11, 64)$
- After MaxPooling2D: $(5, 5, 64)$
- Flatten: 1600
- Dense: $128 \rightarrow 10$

Error Source: Incompatible Shapes

Mismatches between convolution/pooling outputs and Dense inputs are a common source of bugs. Always verify output shapes before Flatten/Dense layers.

How Does Data Flow Through a CNN?

Preprocessing Step is Crucial

Inputs must be scaled (e.g., divided by 255), possibly normalized, and reshaped to match input dimensions expected by the CNN.

Data Preprocessing Checklist

- Resize all images to consistent shape
- Normalize pixel values: $x' = x/255.0$
- One-hot encode class labels (if needed)
- Split data: training, validation, test

How Does Data Flow Through a CNN?

Stage	Transformation Description
Input Preprocessing	Resize, normalize, reshape input tensors
Convolutional Layers	Learn spatial features using shared filters
Pooling Layers	Downsample feature maps to reduce dimensions
Flatten	Convert 3D tensors to 1D vectors
Dense Layers	Combine features for classification or regression
Output Layer	Maps to class scores or predictions

Current Section

1. Introduction: Why CNNs for Image Processing?
2. CNN Architecture: Components and Layer Types
3. Convolution Operation and Parameter Sharing
4. Pooling Layers and Spatial Reduction
5. Activation Functions in CNNs
6. Fully Connected Layers and Output Interpretation
7. Regularization in CNNs: Dropout, L2, BatchNorm
8. Hierarchical Feature Learning in CNNs
9. Comparing CNNs to ANNs
10. CNN Processing Pipeline
11. Real-World Architectures: LeNet, AlexNet, VGG, ResNet, Inception
12. CNN Architecture Implementations

How Are CNNs Used in the Real World?

Let's explore how foundational CNN architectures are designed and applied in practice. Each solved critical limitations of previous models.

Why Study Architectures?

Understanding classic architectures helps us:

- Grasp model design trade-offs
- Choose appropriate depth/complexity
- Leverage pretrained weights in practice

How Are CNNs Used in the Real World?

LeNet-5 (1998) – Digit Recognition Pioneer

- Input: 32×32 grayscale image
- Conv $\rightarrow$ Pool $\rightarrow$ Conv $\rightarrow$ Pool $\rightarrow$ FC $\rightarrow$ Output
- Uses tanh activation, average pooling
- Total params: 60K

How Are CNNs Used in the Real World?

LeNet-5 Style Model in Keras

```
model = Sequential([
    Conv2D(6, (5, 5), activation='tanh', input_shape=(32, 32, 1)),
    AveragePooling2D(),
    Conv2D(16, (5, 5), activation='tanh'),
    AveragePooling2D(),
    Flatten(),
    Dense(120, activation='tanh'),
    Dense(84, activation='tanh'),
    Dense(10, activation='softmax')
])
```

How Are CNNs Used in the Real World?

AlexNet (2012) – Deep Learning Breakthrough

- 8 layers: 5 Conv + 3 FC
- ReLU activations, dropout, data augmentation
- Uses 11x11 and 5x5 kernels early on
- Input: $227 \times 227 \times 3$
- Top-5 error in ImageNet: 15.3%

How Are CNNs Used in the Real World?

VGGNet (2014) – Simplicity via Depth

- Uses only 3×3 convolutions
- Very deep: VGG-16 has 13 conv + 3 FC layers
- Input: $224 \times 224 \times 3$
- Huge model: 138M parameters

Formula Insight: Stacking Two 3×3 Convs $\approx$ One 5×5 Conv

How Are CNNs Used in the Real World?

ResNet (2015) – Skip Connections for Depth

- Introduces identity shortcut (residual) connections:

$$\mathbf{y} = F(\mathbf{x}) + \mathbf{x}$$

- Enables very deep networks (ResNet-50, 101, 152)
- Solves vanishing gradient problem

How Are CNNs Used in the Real World?

Inception (GoogLeNet, 2014â16) – Multi-Scale Feature Fusion

- Uses parallel paths with 1×1, 3×3, and 5×5 filters
- Dimensionality reduction with 1×1 convolutions:

$$\text{output}_{1\times1} = W_{1\times1} * X$$

- Inception-v3, v4 include batch norm, factorized convolutions

Trade-Offs in Real Architectures

- Deeper $\Rightarrow$ better features, harder to train
- Bigger $\Rightarrow$ more capacity, slower inference
- Residuals and bottlenecks balance both

How Are CNNs Used in the Real World?

Practical Deployment Tip

Use pretrained versions of these models (e.g., from Keras applications) and fine-tune for your dataset. Saves time and improves accuracy on limited data.

Architecture	Year	Key Feature	Typical Use Case
LeNet-5	1998	Simple pipeline for digit recognition	MNIST, OCR
AlexNet	2012	ReLU, dropout, GPU training	ImageNet, general vision
VGG-16/19	2014	Deep with small filters	Transfer learning
ResNet-50+	2015	Residual connections for depth	Complex recognition tasks
Inception-v3	2016	Parallel multiscale filters	Real-time, mobile vision

Current Section

1 Introduction: Why CNNs for Image Processing?
2 CNN Architecture: Components and Layer Types
3 Convolution Operation and Parameter Sharing
4 Pooling Layers and Spatial Reduction
5 Activation Functions in CNNs
6 Fully Connected Layers and Output Interpretation
7 Regularization in CNNs: Dropout, L2, BatchNorm
8 Hierarchical Feature Learning in CNNs
9 Comparing CNNs to ANNs
10 CNN Processing Pipeline
11 Real-World Architectures: LeNet, AlexNet, VGG, ResNet, Inception
12 CNN Architecture Implementations

LeNet, AlexNet, VGG, ResNet, and Inception in Code

We now implement the core structure of each major CNN architecture using TensorFlow/Keras. Each example is simplified for clarity and educational purposes.

LeNet-5 (for grayscale 32×32 input)

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, AveragePooling2D,
    Flatten, Dense

lenet = Sequential([
    Conv2D(6, kernel_size=(5, 5), activation='tanh',
        input_shape=(32, 32, 1)),
    AveragePooling2D(),
    Conv2D(16, kernel_size=(5, 5), activation='tanh'),
    AveragePooling2D(),
    Flatten(),
    Dense(120, activation='tanh'),
    Dense(84, activation='tanh'),
    Dense(10, activation='softmax')  # 10-class classification
])
```

LeNet, AlexNet, VGG, ResNet, and Inception in Code

AlexNet (simplified for $227 \times 227 \times 3$ input)

```python
from tensorflow.keras.layers import Dropout, MaxPooling2D
alexnet = Sequential([
    Conv2D(96, kernel_size=(11,11), strides=4, activation='relu',
         input_shape=(227,227,3)),
    MaxPooling2D(pool_size=(3,3), strides=2),
    Conv2D(256, kernel_size=(5,5), padding='same', activation='relu'),
    MaxPooling2D(pool_size=(3,3), strides=2),
    Conv2D(384, kernel_size=(3,3), padding='same', activation='relu'),
    Conv2D(384, kernel_size=(3,3), padding='same', activation='relu'),
    Conv2D(256, kernel_size=(3,3), padding='same', activation='relu'),
    MaxPooling2D(pool_size=(3,3), strides=2),
    Flatten(),
    Dense(4096, activation='relu'),
    Dropout(0.5),
    Dense(4096, activation='relu'),
    Dropout(0.5),
    Dense(1000, activation='softmax')  # ImageNet classes
])
```

LeNet, AlexNet, VGG, ResNet, and Inception in Code

VGG-16 Style (simplified, without loading weights)

```
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Input

input_vgg = Input(shape=(224,224,3))
x = Conv2D(64, (3,3), activation='relu', padding='same')(input_vgg)
x = Conv2D(64, (3,3), activation='relu', padding='same')(x)
x = MaxPooling2D((2,2))(x)

x = Conv2D(128, (3,3), activation='relu', padding='same')(x)
x = Conv2D(128, (3,3), activation='relu', padding='same')(x)
x = MaxPooling2D((2,2))(x)

x = Conv2D(256, (3,3), activation='relu', padding='same')(x)
x = Conv2D(256, (3,3), activation='relu', padding='same')(x)
x = Conv2D(256, (3,3), activation='relu', padding='same')(x)
x = MaxPooling2D((2,2))(x)

x = Conv2D(512, (3,3), activation='relu', padding='same')(x)
x = Conv2D(512, (3,3), activation='relu', padding='same')(x)
x = Conv2D(512, (3,3), activation='relu', padding='same')(x)
x = MaxPooling2D((2,2))(x)

x = Conv2D(512, (3,3), activation='relu', padding='same')(x)
x = Conv2D(512, (3,3), activation='relu', padding='same')(x)
x = Conv2D(512, (3,3), activation='relu', padding='same')(x)
x = MaxPooling2D((2,2))(x)

x = Flatten()(x)
x = Dense(4096, activation='relu')(x)
x = Dense(4096, activation='relu')(x)
output_vgg = Dense(1000, activation='softmax')(x)

vgg = Model(inputs=input_vgg, outputs=output_vgg)
```

LeNet, AlexNet, VGG, ResNet, and Inception in Code

ResNet Block and Model (ResNet-18 style)

```
from tensorflow.keras.layers import Add, BatchNormalization, ReLU

def residual_block(x, filters, downsample=False):
    stride = 2 if downsample else 1
    y = Conv2D(filters, (3, 3), strides=stride, padding='same')(x)
    y = BatchNormalization()(y)
    y = ReLU()(y)
    y = Conv2D(filters, (3, 3), padding='same')(y)
    y = BatchNormalization()(y)

    if downsample:
        x = Conv2D(filters, (1, 1), strides=2)(x)

    return ReLU()(Add()([x, y]))

input_resnet = Input(shape=(224,224,3))
x = Conv2D(64, (7,7), strides=2, padding='same')(input_resnet)
x = MaxPooling2D((3,3), strides=2, padding='same')(x)

x = residual_block(x, 64)
x = residual_block(x, 64)
x = residual_block(x, 128, downsample=True)
x = residual_block(x, 128)

x = GlobalAveragePooling2D()(x)
output_resnet = Dense(1000, activation='softmax')(x)

resnet = Model(inputs=input_resnet, outputs=output_resnet)
```

LeNet, AlexNet, VGG, ResNet, and Inception in Code

Inception Module (Simplified Inception-v1 block)

```
from tensorflow.keras.layers import concatenate

def inception_module(x, f1, f3r, f3, f5r, f5, proj):
    path1 = Conv2D(f1, (1,1), padding='same', activation='relu')(x)
    path2 = Conv2D(f3r, (1,1), activation='relu')(x)
    path2 = Conv2D(f3, (3,3), padding='same', activation='relu')(path2)
    path3 = Conv2D(f5r, (1,1), activation='relu')(x)
    path3 = Conv2D(f5, (5,5), padding='same', activation='relu')(path3)
    path4 = MaxPooling2D((3,3), strides=(1,1), padding='same')(x)
    path4 = Conv2D(proj, (1,1), activation='relu')(path4)
    return concatenate([path1, path2, path3, path4], axis=-1)

input_incept = Input(shape=(224,224,3))
x = Conv2D(64, (7,7), strides=2, padding='same',
     activation='relu')(input_incept)
x = MaxPooling2D((3,3), strides=2, padding='same')(x)

x = inception_module(x, 64, 96, 128, 16, 32, 32)
x = inception_module(x, 128, 128, 192, 32, 96, 64)

x = GlobalAveragePooling2D()(x)
output_incept = Dense(1000, activation='softmax')(x)

inception = Model(inputs=input_incept, outputs=output_incept)
```