Memory updated

Convolutional Neural Networks (CNNs) – The Ultimate Guide

1. Intuitive Understanding of CNNs

Why Do We Need CNNs?

- **Traditional Neural Networks (MLPs) don't scale for images**.
- Images have **spatial structure** (patterns, edges, textures) that **fully connected layers don't preserve**.
- CNNs **reduce computational complexity** by using local receptive fields and shared weights.

Key Idea Behind CNNs

Instead of treating an image as a **flat vector of pixels**, CNNs **preserve spatial relationships** by applying **filters (kernels)** to detect features like:

- **Edges** → Detect shapes
- **Textures** → Identify materials
- **Higher-level patterns** → Recognize objects

2. When to Use and Avoid CNNs

✅ **Use CNNs When:**

- Your data has **spatial hierarchies** (e.g., images, videos).
- You need **translation invariance** (recognizing an object regardless of position).
- The task requires **feature extraction** from raw data.

❌ **Avoid CNNs When:**

- Your data **is not spatially structured** (e.g., tabular data, pure text).
- You have **limited training data** (CNNs require a lot of labeled images).
- The task is **better suited for sequence models** (e.g., RNNs for time series).

---

3. Where CNNs Fail and How to Avoid It

Potential Failures

🚫 **Overfitting on small datasets** → CNNs have a lot of parameters.
🚫 **Not recognizing objects at different scales** → Some CNNs don't handle small vs. large objects well.
🚫 **Ignoring important features due to incorrect filter initialization**.
🚫 **Computational cost** → CNNs require more hardware power than MLPs.

How to Avoid These Issues

✅ **Use data augmentation** (rotation, flipping, cropping).
✅ **Use transfer learning** (pretrained models like ResNet, VGG).
✅ **Add dropout and L2 regularization** to prevent overfitting.
✅ **Use multi-scale architectures** (like Feature Pyramid Networks) for objects at different sizes.

4. Advantages and Disadvantages of CNNs

✅ **Advantages**

- **Extracts local features** efficiently.
- **Parameter sharing** (same filter across different locations reduces computation).
- **Translation invariance** (detects patterns regardless of position).
- **Performs better on image tasks** than MLPs.

❌ **Disadvantages**

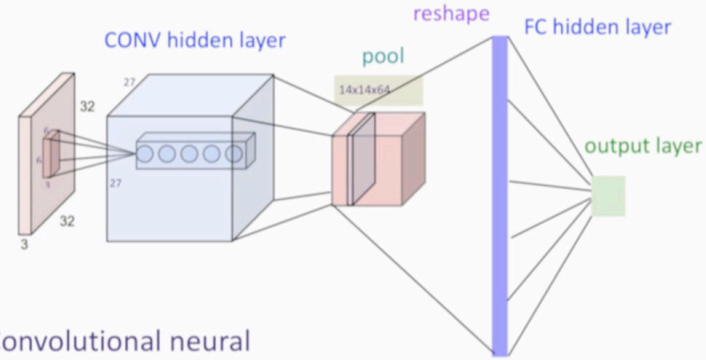
- Computationally expensive.
- Requires large datasets.
- Harder to interpret than decision trees or linear models.

5. Pseudo Code for CNN

plaintext Copy Edit

```
# Step 1: Initialize CNN Parameters
1. Define CNN architecture:
   a. Convolutional layers with filters
   b. Activation functions (ReLU)
   c. Pooling layers (MaxPooling)
   d. Fully connected layers for classification

# Step 2: Training Loop
FOR each epoch:
    1. Forward Propagation:
        a. Apply convolutional layers:
            - Perform convolution (dot product between filters and input)
            - Apply ReLU activation
        b. Apply pooling layers (reduce spatial dimensions)
        c. Flatten output and pass through fully connected layers
        d. Compute final prediction (Softmax for classification)

    2. Compute Loss:
        - Use Cross-Entropy Loss for classification

    3. Backpropagation:
        a. Compute gradient of loss w.r.t. output
        b. Compute gradient of loss w.r.t. filters (using convolution transpose)
        c. Compute gradient of loss w.r.t. fully connected layers
        d. Update weights using Gradient Descent

    4. Print loss and monitor accuracy

# Step 3: Prediction
1. Perform forward propagation on new images.
2. Return predicted class.
```

Training Convolutional Networks



Recall: Convolutional neural networks (CNN) are just regular fully connected (FC) neural networks with some connections removed and some weights shared.

→ **Train with SGD!**

Stanfor

Um, you still have those like backward functions. And in the end, as you said, our conv layer is just a linear layer.

2. Choosing a CNN Architecture

When designing a CNN, you must decide:

- **Number of convolutional layers** (How many filters to extract patterns?)
- **Filter size & number of filters** (How detailed should the feature maps be?)
- **Stride and padding** (How much should the feature map shrink?)
- **Number of pooling layers** (How much spatial reduction do we need?)
- **Fully connected layers** (How complex is the final decision-making step?)

A Common CNN Architecture Example

Layer Type	Purpose	Example Parameters
Input Layer	Takes the raw image	(e.g., 32x32x3 for RGB images)
Conv Layer 1	Detects basic features (edges)	8 filters, size 3x3
ReLU Activation	Introduces non-linearity	f(x) = max(0, x)
Pooling Layer	Reduces spatial dimensions	Max Pooling, 2x2
Conv Layer 2	Detects more complex features (textures)	16 filters, size 3x3
ReLU Activation	Non-linearity	f(x) = max(0, x)
Pooling Layer	Further reduces spatial size	Max Pooling, 2x2
Fully Connected Layer	Final decision-making	128 neurons
Output Layer	Produces classification probabilities	Softmax for multi-class

3. Understanding Each Component of a CNN

Let's break down each part:

3.1 Convolutional Layers (Feature Extraction)

💡 **What it does:**

- Detects features **by applying filters (kernels)** over the input image.
- Each filter moves across the image, computing a **dot product** with the region it overlaps.
- Produces a **feature map**, where bright pixels represent strong activation of a pattern.

💡 **Key Choices in Convolutions:**

Hyperparameter	Effect
Filter Size (e.g., 3x3, 5x5)	Larger filters detect bigger patterns, but lose fine details
Number of Filters	More filters capture more features, but increase parameters
Stride (e.g., 1, 2)	Higher stride means lower resolution output (reduces computation)
Padding (Same/Valid)	"Same" keeps original size, "Valid" shrinks the image

💡 **Mathematics of Convolution:**

A **3x3 filter** on a **5x5 image** without padding produces an **(3x3) output** using:

$$\text{Output Size} = \frac{\text{Input Size} - \text{Filter Size}}{\text{Stride}} + 1$$

A 1×1 **convolution** with **stride = 1** and **padding = 0** operates as follows:

- The **input shape** is (n_h, n_w, n_c), where:
  - n_h = height
  - n_w = width
  - n_c = number of channels
- The **filter size** is 1×1, meaning it applies a separate linear transformation to each spatial location without mixing neighboring pixels.
- **Stride = 1** means the filter moves one step at a time.
- **Padding = 0** means no extra pixels are added, so the height and width remain unchanged.

Output Shape Calculation:

The general formula for the output dimensions in a convolution operation is:

$$\text{output height} = \frac{n_h + 2p - f_h}{s} + 1$$

$$\text{output width} = \frac{n_w + 2p - f_w}{s} + 1$$

where:

- f_h, f_w = filter height and width
- s = stride
- p = padding

Substituting $f_h = 1$, $f_w = 1$, $s = 1$, and $p = 0$:

$$\text{output height} = \frac{n_h + 0 - 1}{1} + 1 = n_h$$

$$\text{output width} = \frac{n_w + 0 - 1}{1} + 1 = n_w$$

Thus, **the height n_h and width n_w remain unchanged.**

3.2 Activation Functions (ReLU, Leaky ReLU)

💡 **What it does:**

- Introduces **non-linearity**, so the network can learn complex patterns.
- Most common: **ReLU (Rectified Linear Unit)**

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

💡 **Effect of Activation Functions:**

Function	When to Use	Why?
ReLU	Default for CNNs	Avoids vanishing gradients
Leaky ReLU	When neurons "die" in ReLU	Allows small negative values
Sigmoid/Tanh	Not recommended	Cause vanishing gradients

3.3 Pooling Layers (Downsampling)

💡 **What it does:**

- **Reduces spatial dimensions**, making the network computationally efficient.
- **Common types:**
  - **Max Pooling**: Takes the highest value in a region (preserves strongest signals)
  - **Average Pooling**: Takes the average value in a region (smoothes output)

💡 **Effect of Pooling:**

Pooling Type	Effect
Max Pooling (2x2)	Reduces size by 75% (selects strongest features)
Average Pooling (2x2)	Reduces size, smooths feature maps

$$\text{Output Size} = \frac{\text{Input Size}}{\text{Pool Size}}$$

3.4 Fully Connected Layers (Final Classification)

💡 **What it does:**

- Converts extracted features into a decision.
- Uses **fully connected (dense) layers** like in MLPs.
- Typically ends with **Softmax activation** for multi-class classification:

$$P(y_i) = \frac{e^{z_i}}{\sum e^{z_j}}$$

How to Calculate the Total Number of Parameters in a CNN

To determine the total number of parameters in a CNN, we need to calculate the **trainable parameters** in each layer:

1. **Convolutional Layers**
2. **Pooling Layers** (no trainable parameters)
3. **Fully Connected (Dense) Layers**

Let's break this down step by step.

---

1. Convolutional Layer Parameters

Each convolutional layer has **trainable parameters**:

- **Filters (Kernels):** Each filter has dimensions:

$$\text{Filter Size} = (H_f \times W_f \times D_{in})$$

  where:

  - H_f, W_f → Filter height & width
  - D_{in} → Number of input channels

- **Number of Filters:** Each filter has its own weights, so we multiply by the **number of filters**:

$$\text{Total Weights} = (H_f \times W_f \times D_{in}) \times D_{out}$$

  where D_{out} is the number of filters.

- **Bias Terms:** Each filter has **one bias value**, so we add:

$$\text{Total Biases} = D_{out}$$

- **Total Parameters per Convolutional Layer:**

$$\text{Total Params} = (H_f \times W_f \times D_{in}) \times D_{out} + D_{out}$$

Example:

- Input: **32×32×3** image
- Filters: **8 filters, each of size 3×3×3**
- Formula:

$$(3 \times 3 \times 3) \times 8 + 8 = (27 \times 8) + 8 = 216 + 8 = 224 \text{ parameters}$$

2. Pooling Layer Parameters

Pooling layers **don't have trainable parameters**, only reduce spatial dimensions.

✅ **Total parameters = 0**

---

3. Fully Connected (Dense) Layer Parameters

A fully connected layer connects **each neuron to every neuron in the previous layer**.

- **Number of Weights:**

$$\text{Total Weights} = \text{Number of Inputs} \times \text{Number of Outputs}$$

- **Number of Biases:**

$$\text{Total Biases} = \text{Number of Outputs}$$

- **Total Parameters in a Dense Layer:**

$$\text{Total Params} = (\text{Inputs} \times \text{Outputs}) + \text{Outputs}$$

Example:

- Flattened input from a **pooled feature map**: **8 × 14 × 14** (size from convolution)
- Fully connected layer with **128 neurons**:

$$(8 \times 14 \times 14) \times 128 + 128$$

$$= (1568 \times 128) + 128 = 200704 + 128 = 200832 \text{ parameters}$$

---

4. Output Layer Parameters

The output layer maps the last fully connected layer to the **final classes**.

- **Weights:** $(\text{Last FC Neurons} \times \text{Classes})$
- **Biases:** Classes
- **Total:**

$$(\text{Last FC Neurons} \times \text{Classes}) + \text{Classes}$$

Example (10 classes for MNIST):

- **Previous layer: 128 neurons**
- **Formula:**

$$(128 \times 10) + 10 = 1280 + 10 = 1290$$

5. Total Parameters in a CNN

Adding all layers together:

Layer Type	Parameters Calculation	Total Parameters
Conv Layer 1 (3x3, 3 channels, 8 filters)	$(3 \times 3 \times 3) \times 8 + 8$	**224**
Pooling Layer 1	No parameters	**0**
Conv Layer 2 (3x3, 8 channels, 16 filters)	$(3 \times 3 \times 8) \times 16 + 16$	**1168**
Pooling Layer 2	No parameters	**0**
Fully Connected (FC, 8×14×14 → 128 neurons)	$(8 \times 14 \times 14) \times 128 + 128$	**200832**
Output Layer (128 → 10 classes)	$(128 \times 10) + 10$	**1290**
Total CNN Parameters	Sum of all layers	**203514**

Final Summary

- 🔷 **Conv Layers**: Parameters depend on **filter size, input depth, and number of filters**.
- 🔷 **Pooling Layers**: No parameters, just **reduce dimensions**.
- 🔷 **Fully Connected Layers**: Parameters scale with **number of neurons**.
- 🔷 **Total Parameters**: Sum of all weight and bias parameters.

Memory updated

Fully Commented Manual Implementation of CNN (From Scratch)

```
119
120         # Update weights and biases
121         self.fc_weights -= self.learning_rate * dW_fc
122         self.fc_bias -= self.learning_rate * db_fc
123         self.filters -= self.learning_rate * dConv
124
125     def train(self, X, y, epochs=10):
126         """Trains the CNN using gradient descent."""
127         for epoch in range(epochs):
128             output = self.forward(X)
129             loss = self.compute_loss(y)
130             self.backward(X, y)
131
132             if epoch % 1 == 0:
133                 print(f"Epoch {epoch+1}, Loss: {loss:.4f}")
134
135     def predict(self, X):
136         """Predicts the class of an image."""
137         output = self.forward(X)
138         return np.argmax(output)  # Return class with highest probability
139 ⌘K for Command, ⌘L for Cascade
```

```python
    def compute_loss(self, y_true):
        """Computes cross-entropy loss."""
        return -np.log(self.softmax_output[y_true, 0])

    def backward(self, X, y_true):
        """Performs backpropagation to compute gradients and update weights."""
        # Compute gradient of softmax loss
        dZ = self.softmax_output
        dZ[y_true, 0] -= 1  # Derivative of softmax loss

        # Compute gradients for fully connected layer
        dW_fc = np.dot(dZ, self.flattened.T)
        db_fc = dZ

        # Backpropagate to pooling layer
        dFlatten = np.dot(self.fc_weights.T, dZ).reshape(self.pooled_output.shape)

        # Backpropagate through max pooling (assign gradients to max locations)
        dPool = np.zeros_like(self.pooled_output)
        h, w, d = dPool.shape
        for i in range(h):
            for j in range(w):
                region = self.relu_output[i*self.pool_size:(i+1)*self.pool_size, j*self.pool_size:(j+1)*self.pool_size, :]
                max_values = np.max(region, axis=(0, 1), keepdims=True)
                dPool[i, j] = (region == max_values) * dFlatten[i, j]

        # Backpropagate through ReLU
        dReLU = dPool * (self.conv_output > 0)

        # Backpropagate through convolution
        dConv = np.zeros_like(self.filters)
        input_h, input_w = X.shape
        for f in range(self.num_filters):
            for i in range(input_h - self.filter_size + 1):
                for j in range(input_w - self.filter_size + 1):
                    region = X[i:i+self.filter_size, j:j+self.filter_size]
                    dConv[f] += region * dReLU[i, j, f]
```

```python
    def convolve(self, X):
        """Performs convolution operation."""
        h, w = X.shape
        output_dim = h - self.filter_size + 1  # Output size after convolution
        conv_output = np.zeros((output_dim, output_dim, self.num_filters))

        # Apply each filter to the image
        for f in range(self.num_filters):
            for i in range(output_dim):
                for j in range(output_dim):
                    region = X[i:i+self.filter_size, j:j+self.filter_size]  # Extract region
                    conv_output[i, j, f] = np.sum(region * self.filters[f])  # Apply filter

        return conv_output

    def pool(self, X):
        """Performs 2x2 max pooling."""
        h, w, d = X.shape
        new_h, new_w = h // self.pool_size, w // self.pool_size
        pooled_output = np.zeros((new_h, new_w, d))

        for i in range(new_h):
            for j in range(new_w):
                region = X[i*self.pool_size:(i+1)*self.pool_size, j*self.pool_size:(j+1)*self.pool_size, :]
                pooled_output[i, j] = np.max(region, axis=(0, 1))  # Max pooling

        return pooled_output

    def forward(self, X):
        """Performs forward propagation through the network."""
        self.conv_output = self.convolve(X)  # Convolution step
        self.relu_output = self.relu(self.conv_output)  # Activation function
        self.pooled_output = self.pool(self.relu_output)  # Pooling operation
        self.flattened = self.pooled_output.flatten().reshape(-1, 1)  # Flatten feature map
        self.fc_output = np.dot(self.fc_weights, self.flattened) + self.fc_bias  # Fully connected layer
        self.softmax_output = self.softmax(self.fc_output)  # Softmax activation
        return self.softmax_output
```

```python
import numpy as np

class SimpleCNN:
    """
    A basic Convolutional Neural Network (CNN) with one convolutional layer,
    one pooling layer, and a fully connected output layer.
    """

    def __init__(self, num_filters=8, filter_size=3, input_size=(28, 28),
                 pool_size=2, output_classes=10, learning_rate=0.01):
        """
        Initialize CNN parameters.
        :param num_filters: Number of filters in the convolutional layer.
        :param filter_size: Size of each filter (kernel).
        :param input_size: Dimensions of input images (height, width).
        :param pool_size: Size of pooling window.
        :param output_classes: Number of output classes (for classification).
        :param learning_rate: Step size for weight updates.
        """
        self.num_filters = num_filters
        self.filter_size = filter_size
        self.input_size = input_size
        self.pool_size = pool_size
        self.output_classes = output_classes
        self.learning_rate = learning_rate

        # Initialize filters randomly with small values
        self.filters = np.random.randn(num_filters, filter_size, filter_size) / 9

        # Fully connected layer weights (random initialization)
        fc_input_size = (input_size[0] // pool_size) * (input_size[1] // pool_size) * num_filters
        self.fc_weights = np.random.randn(output_classes, fc_input_size)
        self.fc_bias = np.zeros((output_classes, 1))

    def relu(self, x):
        """Applies ReLU activation function."""
        return np.maximum(0, x)

    def softmax(self, x):
        """Applies the softmax activation function to convert scores into probabilities."""
        exp_x = np.exp(x - np.max(x))  # Subtract max for numerical stability
        return exp_x / np.sum(exp_x, axis=0, keepdims=True)
```

Alternative: Scikit-Learn Compatible CNN Using PyTorch

If you want a CNN **inside a Scikit-Learn-like API**, use **Torch's** `skorch`:

python Copy Edit

```python
from skorch import NeuralNetClassifier
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torchvision import datasets, transforms
from torch.utils.data import DataLoader

# Define a simple CNN using PyTorch
class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        self.conv1 = nn.Conv2d(1, 8, kernel_size=3, stride=1, padding=1)
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2)
        self.fc1 = nn.Linear(8 * 14 * 14, 10)  # Assuming 28x28 input images

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = x.view(-1, 8 * 14 * 14)  # Flatten
        x = self.fc1(x)
        return F.log_softmax(x, dim=1)

# Wrap in a Scikit-Learn-like classifier
cnn = NeuralNetClassifier(
    CNN,
    criterion=nn.CrossEntropyLoss,
    optimizer=optim.Adam,
    max_epochs=10,
    lr=0.001,
)

# Train on MNIST dataset
mnist = datasets.MNIST(root='./data', train=True, transform=transforms.ToTensor(), d
cnn.fit(DataLoader(mnist, batch_size=32, ...nist.targets)
```