CNN

🧩 Basic Building Blocks of CNN Architecture

---

1. Input Layer

Concept:

- This is the layer where we feed the raw data (e.g., an image) into the CNN.
- Images are usually represented as **3D tensors**:

  Height×Width×Channels

Examples:

- Grayscale image → 28×28×1
- RGB image → 224×224×3

Role:

- Acts as the entry point.
- Normalization (e.g., scaling pixel values between 0–1 or -1–1) often happens here to stabilize training.

---

2. Convolutional Layer

Concept:

- The **heart of CNNs**.
- Uses **convolutional kernels (filters)** to scan across the input and extract features.

🔷 Padding in CNNs

What is Padding?

- **Padding** means adding extra pixels (usually zeros) around the **border of the input image** before applying convolution.
- Without padding, the kernel **shrinks** the output size after each convolution

Why Do We Need Padding?

a) To Control Output Size - Every convolution reduces the image size (unless padding is used).

b) To Preserve Edge Information - Without padding, edge pixels are used **less frequently** → information loss.
c) To Allow "Same" Convolution - **Same padding:** Output size = Input size (when stride = 1).

Types of Padding:

- **Zero Padding** (most common): Add zeros around the image.
- **Reflect Padding:** Mirror pixels at the edge.
- **Replicate Padding:** Repeat the edge value.
- **Circular Padding:** Wrap around like a torus.

Output Size Formula (with Padding)

For input size n, kernel size f, stride s, and padding p:

$$\text{Output size} = \frac{(n + 2p - f)}{s} + 1$$

🔷 Convolutional Kernel

What is a Kernel?

- A **kernel** (also called a filter) is a **small matrix of learnable weights**.
- Typical sizes: 3×3, 5×5, 7×7.

- It slides (convolves) across the image, performing a **dot product** between the kernel and the local image region.

Example:

If the input image patch =

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

and kernel =

$$\begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix}$$

Then convolution = sum of element-wise multiplication → detects **vertical edges**.

Stride, Kernel Size, and Dilation in CNNs

1. **Kernel Size**

   Definition:

   - The **dimensions (height × width)** of the filter (kernel).
   - Common choices: 3 × 3, 5 × 5, 7 × 7.

   Effect:

   - Determines how **much of the image the kernel looks at once** (the receptive field).
   - **Small kernel (3×3):** Focuses on fine details (edges, textures).
   - **Large kernel (7×7 or 11×11):** Captures more global context but has more parameters.

2. **Stride**

   Definition:

     - The **step size** with which the kernel moves across the image.

Effect:

- Controls **output size** (downsampling).

📌 Example:

- Stride = 1 → kernel moves 1 pixel at a time (high resolution).
- Stride = 2 → kernel moves 2 pixels at a time (reduces output size).

Formula for Output Size (no padding):

If input = n × n, kernel = f × f, stride = s:

$$\text{Output size} = \frac{(n-f)}{s} + 1$$

👉 Higher stride = smaller output (fewer computations, less detail).

3. **Dilation**

Definition:

- Dilation introduces **spaces (gaps) between kernel elements**.
- Kernel elements are spread out instead of being adjacent.

Effect:

- Expands the **receptive field** without increasing kernel size.
- Helps capture **larger context** while keeping parameter count small.

📌 Example:

- Kernel size = 3 × 3, dilation = 1 → normal kernel:

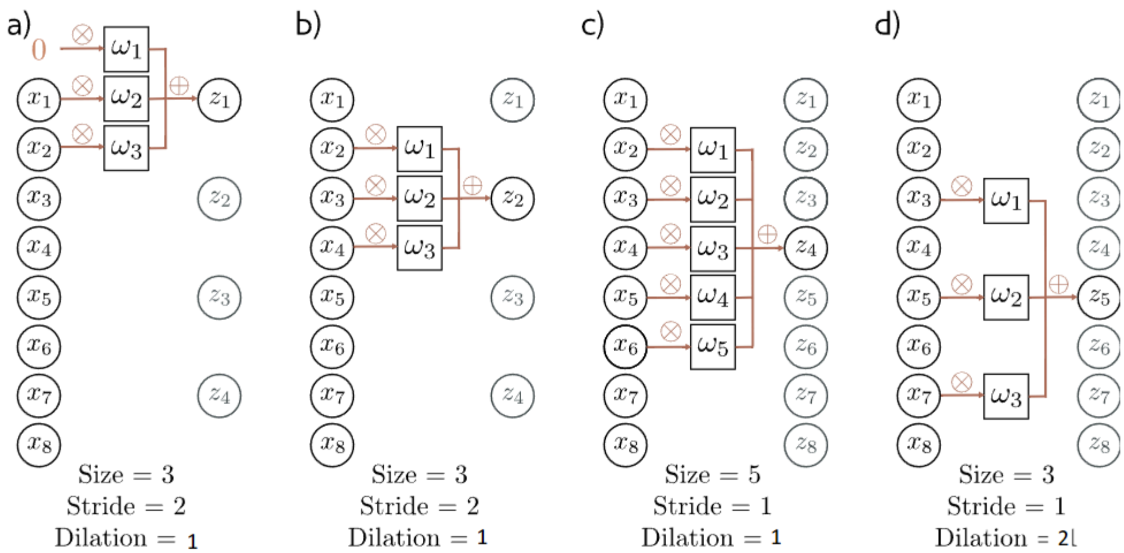
$$\begin{bmatrix} * & * & * \\ * & * & * \\ * & * & * \end{bmatrix}$$

- Kernel size = 3 × 3, dilation = 2 → spaces added:

$$\begin{bmatrix} * & 0 & * & 0 & * \\ 0 & 0 & 0 & 0 & 0 \\ * & 0 & * & 0 & * \\ 0 & 0 & 0 & 0 & 0 \\ * & 0 & * & 0 & * \end{bmatrix}$$

(where `*` = kernel element, `0` = skipped pixel).

👉 Used in **dilated convolutions** (common in segmentation models like DeepLab).



Why is the Kernel Important?

👉 Think of the kernel as a **feature detector**.

- **Different kernels learn different features:**
  - One kernel may detect **horizontal edges**.
  - Another may detect **vertical edges**.
  - Another may detect **textures, corners, or color blobs**.
- Instead of manually designing features (like in old computer vision), CNNs **learn the best kernels automatically** during training.

Significance of Kernels

🔷 a) **Local Feature Extraction**

- Instead of looking at the entire image at once, kernels focus on **local patterns** (small regions).
- This is biologically inspired → just like the human visual cortex responds to local edges and textures.

🔷 b) **Parameter Efficiency**

- A kernel is small (say 3×3) but **reused across the whole image**.
- This **weight sharing** means:
  - Fewer parameters than fully connected layers.
  - More efficient training.

🔷 c) **Translation Invariance**

- If a cat is in the top-left or bottom-right, the **same kernel** can detect its fur/whiskers.
- Kernels make CNNs **robust to position changes**.

🔷 d) **Hierarchical Learning**

- **First layers' kernels**: detect simple patterns (edges, lines).
- **Middle layers' kernels**: detect shapes (eyes, ears, wheels).
- **Deeper layers' kernels**: detect high-level concepts (faces, cars, cats).

Intuitive Analogy

- Imagine looking at a picture with a **magnifying glass**.
- Each kernel is like a different magnifying glass lens:
  - One lens highlights edges.
  - Another highlights textures.
  - Another highlights colors.
- Together, they build a **complete understanding** of the image.

Summary

- **Kernel = feature detector.**
- It is a **small matrix of learnable weights**.
- Scans the image to extract **local patterns**.

- Enables CNNs to be efficient, translation-invariant, and powerful at hierarchical feature learning.

> **Why Kernels Are Odd-Sized (3×3, 5×5, ...)?**

- Odd-sized kernels have a **clear center pixel (Center Pixel Symmetry)**, better symmetry, and easier padding.
- They provide more stable feature detection and alignment.
- **3×3 kernels** are the most popular: small, efficient, and stackable to approximate larger receptive fields.
- Even-sized kernels (2×2, 4×4) exist but are rare, mainly in **pooling layers** or niche cases.

Convolution Operations in CNNs

1. 2D Convolution

Concept:

- Standard convolution in CNNs, where a 2D **kernel (filter)** slides over a 2D input (image).
- Performs **dot product** between the kernel and local image regions.

Formula for Output Size:

$$\text{Output size} = \frac{(n + 2p - f)}{s} + 1$$

where n=input size, p=padding, f=kernel size, s=stride.

---

2. Channels in 2D Convolution

- Real images are not just grayscale; they often have **3 channels (RGB)**.
- A kernel spans **all channels of the input**.

📌 Example:

- Input = 32×32×3 (RGB)

- Kernel = 3×3×3 (covers all channels)
- One kernel produces **one 2D feature map**.
- If we use 64 kernels → we get 64 feature maps → output size = 32×32×64

---

3. How Many Parameters?

For each kernel:

$$\text{Parameters} = (f \times f \times c) + 1$$

where:

- f = kernel size
- c = input channels
- +1 = bias term

📌 Example:

- Input = RGB (3 channels)
- Kernel = 3 × 3
- Parameters per kernel = 3 × 3 × 3 + 1 = 28
- If 64 kernels → 28 × 64 = 1792 parameters.

---

4. Different Types of Convolution

CNNs extend the basic convolution idea in multiple ways:

🔷 a) **Dilated (Atrous) Convolution**

- Introduces **gaps** between kernel elements.
- Expands **receptive field** without increasing kernel size.

📌 Example:

- Normal 3×3 kernel → covers 9 pixels.
- Dilation=2 → kernel skips pixels → effectively covers 5×5 area.

Use: Semantic segmentation (captures global context).

🔷 b) **Spatially Separable Convolution**

- Breaks a 2D kernel into **two 1D kernels**.
- Example: 3×3 kernel → do 3×1 then 1×3.

Advantage:

- Fewer parameters (9 → 6).
- Faster computation.

🔷 c) **Depthwise Separable Convolution**

- Splits the convolution into **two steps**:
    1. **Depthwise convolution:** Apply one kernel per channel independently.
    2. **Pointwise convolution (1×1):** Combine across channels.

📌 Example:

- Normal convolution with 3×3×3 kernel = 27 params.
- Depthwise separable:
    - Depthwise (3×3×1 each channel) = 9 params × 3 = 27
    - Pointwise (1×1×3) = 3 params
    - Total = 30 vs normal convolution with many filters.

Use: MobileNets, efficient CNNs.

🔷 d) **Transposed Convolution (Deconvolution)**

- "Inverse" of convolution.
- Instead of downsampling, it **upsamples** (increases spatial resolution).
- Used in:
    - Image generation (GANs).
    - Segmentation (UNet, autoencoders).

📌 Example:

- Input = 4×4

- Apply transposed convolution with stride=2 → Output = 8×8.

---

3. Pooling Layer

Concept:

- Reduces the **spatial dimensions** (height, width) while keeping important features.
- Makes the model more **computationally efficient** and **translation-invariant**.

Types:

1. **Max Pooling:** Takes the maximum value in each region (common).
    - Example: From a 2×2 block → pick the max value.
2. **Average Pooling:** Takes the average of the values.
3. **Global Pooling:** Reduces the entire feature map to a single value.

Role:

- Summarizes features.
- Prevents overfitting by reducing complexity.

---

4. Non-Linearity (Activation Function)

Concept:

- After convolution, the output is still **linear**.
- We need **non-linearity** to let the network learn complex patterns.

Common Activations:

- **ReLU (Rectified Linear Unit):**

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

    - Fast and prevents vanishing gradients.
- **Sigmoid:** Squashes values to (0, 1). Used in binary classification.
- **Tanh:** Squashes values to (-1, 1).

- **Softmax:** Turns outputs into probabilities for multi-class classification.

Role:

- Makes CNNs capable of learning **non-linear decision boundaries**.

---

5. Fully Connected Layer (Dense Layer)

Concept:

- After multiple convolution + pooling layers, we flatten the feature maps into a **1D vector**.
- This vector is fed into fully connected (dense) layers.

Role:

- Combines extracted features to make the final decision (classification, regression, etc.).
- Similar to a traditional **neural network layer**.

Example:

- Image (cat/dog) → convolution layers extract "fur", "whiskers", "ears" → fully connected layer combines and outputs:
    - Cat: 0.95
    - Dog: 0.05

---

6. Loss Layer

Concept:

- Measures how far the network's predictions are from the true labels.
- Provides a **feedback signal** for training (via backpropagation).

Common Loss Functions:

- **Cross-Entropy Loss:**
    - For classification (e.g., cat vs dog).

- Encourages predicted probability distribution to match true labels.
- **Mean Squared Error (MSE):**
  - For regression tasks.
- **Categorical Cross-Entropy:**
  - For multi-class classification.

Role:

- Guides weight updates during **training**.
- The optimizer (e.g., SGD, Adam) minimizes this loss.

🔗 Putting It All Together (Flow)

1. **Input Layer** → Raw image (e.g., 224×224×3).
2. **Convolutional Layer** → Extracts local patterns (edges, shapes).
3. **Activation (Non-Linearity)** → Adds complexity (ReLU).
4. **Pooling Layer** → Downsamples features (max pooling).
5. Repeat Conv + Activation + Pool layers multiple times.
6. **Flatten + Fully Connected Layer** → Combines features for classification.
7. **Loss Layer** → Compares prediction vs truth and updates weights.

✅ This is the **core pipeline of CNNs**.

Batch Normalization

Batch Normalization is a technique to **normalize the activations (inputs) of each layer** so that training becomes faster, more stable, and less sensitive to initialization.

It was introduced by **Ioffe & Szegedy (2015)**.

🔷 Why do we need it?

- During training, the distribution of activations inside the network keeps changing as the weights update.
- This phenomenon is called **Internal Covariate Shift**.
- Because of this, each layer has to keep adapting to changing inputs, making training **slow and unstable**.
- Batch Normalization reduces this problem by **normalizing the inputs of each layer** to a stable distribution.

🔷 How does it work? (Step-by-Step)

Given an input mini-batch $x = \{x_1, x_2, \ldots, x_m\}$:

1. **Compute the batch mean & variance**

$$\mu_B = \frac{1}{m} \sum_{i=1}^{m} x_i$$

$$\sigma_B^2 = \frac{1}{m} \sum_{i=1}^{m} (x_i - \mu_B)^2$$

2. **Normalize** each activation

$$\hat{x}_i = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$$

   (ϵ is a small constant for numerical stability).

3. **Scale and shift (learnable parameters)**

   $y_i = \gamma \hat{x}_i + \beta$

   where:

   - γ = learnable scale factor
   - β = learnable shift factor

🔷 Intuition

- Normalization ensures that activations don't explode or vanish.
- γ,β allow the model to "undo" normalization if needed — so BN doesn't limit the network's capacity.

🔷 Where is BN applied?

- Usually applied **after a convolution / fully connected layer, before activation** (though some frameworks use it after activation).

🔷 Benefits

✅ Faster convergence (reduces training time).

✅ Allows higher learning rates (less risk of divergence).

✅ Acts as a regularizer (sometimes reducing the need for dropout).

✅ Reduces sensitivity to initialization.

🔷 At Inference Time

- During testing, we don't compute batch statistics.
- Instead, we use **running averages of mean and variance** computed during training.

👉 So in your CNN pipeline:

Conv → BatchNorm → Activation (ReLU) → Pooling → … → Fully Connected

---

Layer Normalization

Layer Normalization (LayerNorm) is a normalization technique where we normalize across the **features of a single training example (per layer)** instead of across the **batch**.

It was introduced by **Ba, Kiros, and Hinton (2016)**.

🔷 Why LayerNorm?

- **BatchNorm works poorly when batch size is very small** (because mean/variance estimates become noisy).
- In **RNNs, Transformers**, or sequence models, the notion of "batch statistics" is not always well-defined.
- LayerNorm fixes this by **normalizing per sample, across its hidden units**, independent of batch size.

🔷 How it works (Step-by-step)

Suppose you have a layer output for **one training sample**:

$x = (x_1, x_2, \ldots, x_H)$

where H is the number of hidden units (features).

1. **Compute mean across features** (not across batch):

$$\mu = \frac{1}{H} \sum_{j=1}^{H} x_j$$

2. **Compute variance across features**:

$$\sigma^2 = \frac{1}{H} \sum_{j=1}^{H} (x_j - \mu)^2$$

3. **Normalize each feature**:

$$\hat{x}_j = \frac{x_j - \mu}{\sqrt{\sigma^2 + \epsilon}}$$

4. **Scale and shift (learnable parameters, per feature)**:

$$y_j = \gamma_j \hat{x}_j + \beta_j$$

🔷 Key Difference (BN vs LN)

Aspect	Batch Normalization (BN)	Layer Normalization (LN)
Normalizes over	Batch dimension (all samples in batch)	Feature dimension (per sample)
Depends on batch size	✅ Yes (large batch needed)	❌ No (works even with batch=1)
Popular in	CNNs (vision tasks)	RNNs, Transformers (NLP tasks)
During inference	Uses running mean/variance	No need for running averages

---

🔷 Intuition

- **BatchNorm:** "Let's make sure all examples in the mini-batch have activations with similar distribution."
- **LayerNorm:** "Let's make sure all features of a single example are balanced and stable."

🔷 Example Use Cases

- **BatchNorm**: Image classification CNNs (ResNet, VGG, etc.).
- **LayerNorm**: Transformers (BERT, GPT), RNNs, seq2seq models.

✅ So:

- For **images**, BatchNorm is king.
- For **text/sequence models**, LayerNorm dominates.

Model Regularization

Model regularization refers to a set of techniques that prevent a model from **overfitting** the training data, ensuring it **generalizes well** to unseen data.

👉 In short: **Regularization = Controlling model complexity.**

🔷 Why do we need Regularization?

- Deep neural networks have millions of parameters → they can **memorize training data** instead of learning patterns.
- Without regularization → high training accuracy but poor test accuracy.

Regularization techniques add **constraints or penalties** to prevent the model from being overly complex.

🔷 Types of Regularization

1. L1 and L2 Regularization (Weight Penalties)

- Add penalty term to the loss function.
- **L1 (Lasso):**

$$L = L_{\text{original}} + \lambda \sum |w_i|$$

  → Encourages sparsity (some weights become 0).

- **L2 (Ridge):**

$$L = L_{\text{original}} + \lambda \sum w_i^2$$

  → Keeps weights small, distributes importance evenly.

---

2. Dropout

- During training, randomly "drop" (set to 0) some neurons.
- Prevents co-adaptation of neurons.
- Example: Dropout = 0.5 → half neurons are inactive each forward pass.
- At test time, all neurons are active but scaled.

---

3. Early Stopping

- Monitor validation loss.

- Stop training **before overfitting starts** (when validation loss increases while training loss keeps decreasing).

---

4. Data Augmentation

- Artificially expand dataset: rotations, flips, color shifts (images), synonym replacement (text).
- Helps model see diverse examples → reduces overfitting.

---

5. Batch Normalization / Layer Normalization

- They act like implicit regularizers by stabilizing training and reducing internal covariate shift.

---

6. Weight Constraints

- Restrict weights within a range (e.g., max-norm regularization).

🔷 Intuition

- Imagine you're fitting a curve to data points:
  - **No regularization** → overly wiggly curve (memorization).
  - **With regularization** → smoother curve (generalization).

🔷 Summary

- **Model regularization = Prevent overfitting.**
- **Techniques:** L1/L2 penalties, Dropout, Early stopping, Data augmentation, Weight constraints.
- In practice, we usually combine several (e.g., L2 + Dropout + Early stopping).
