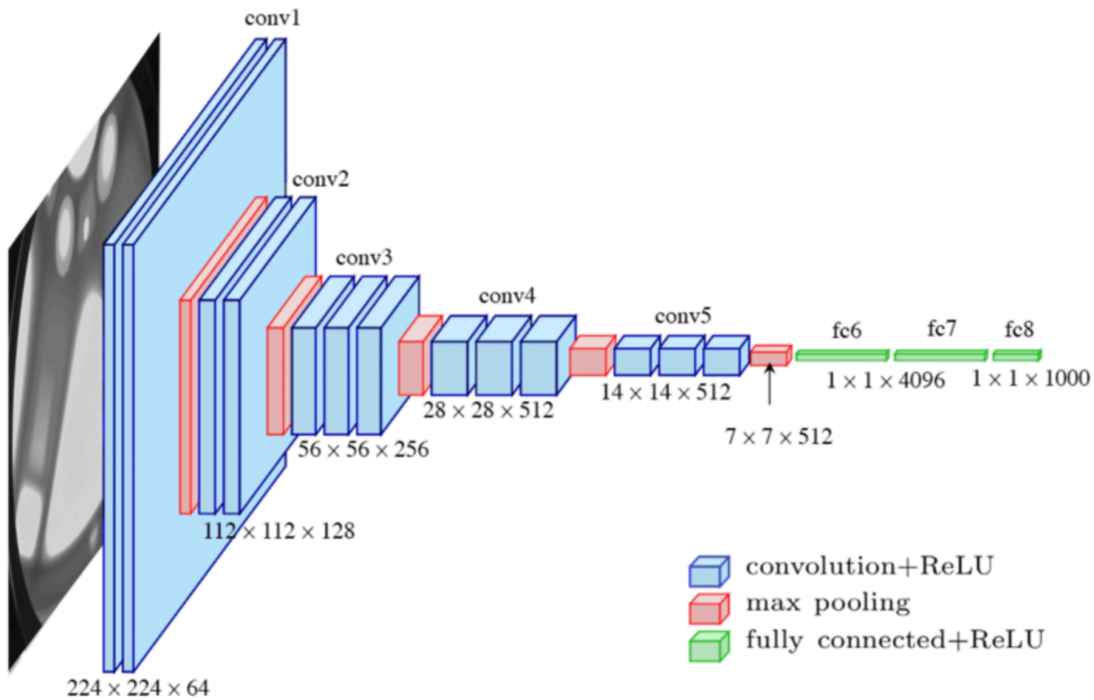
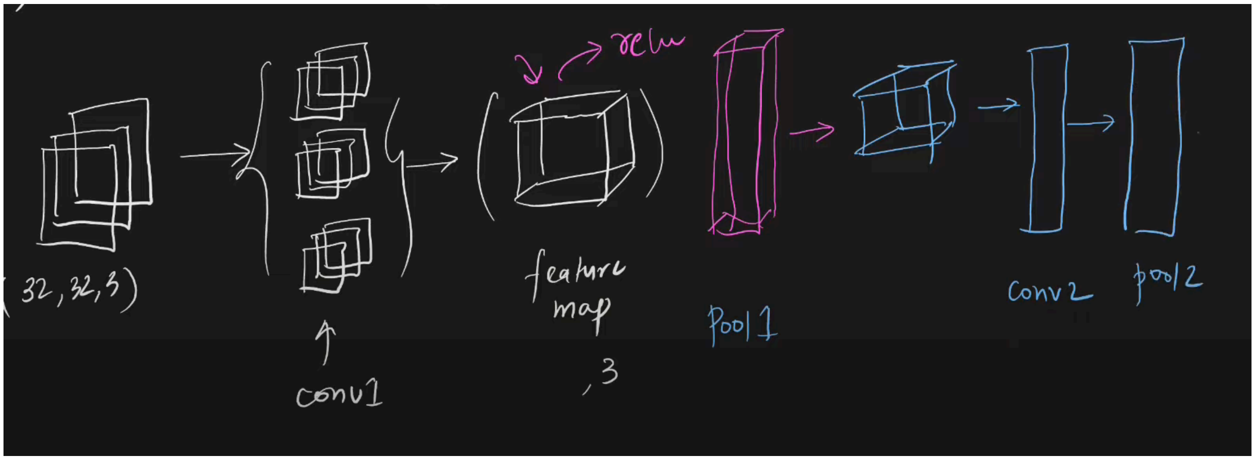
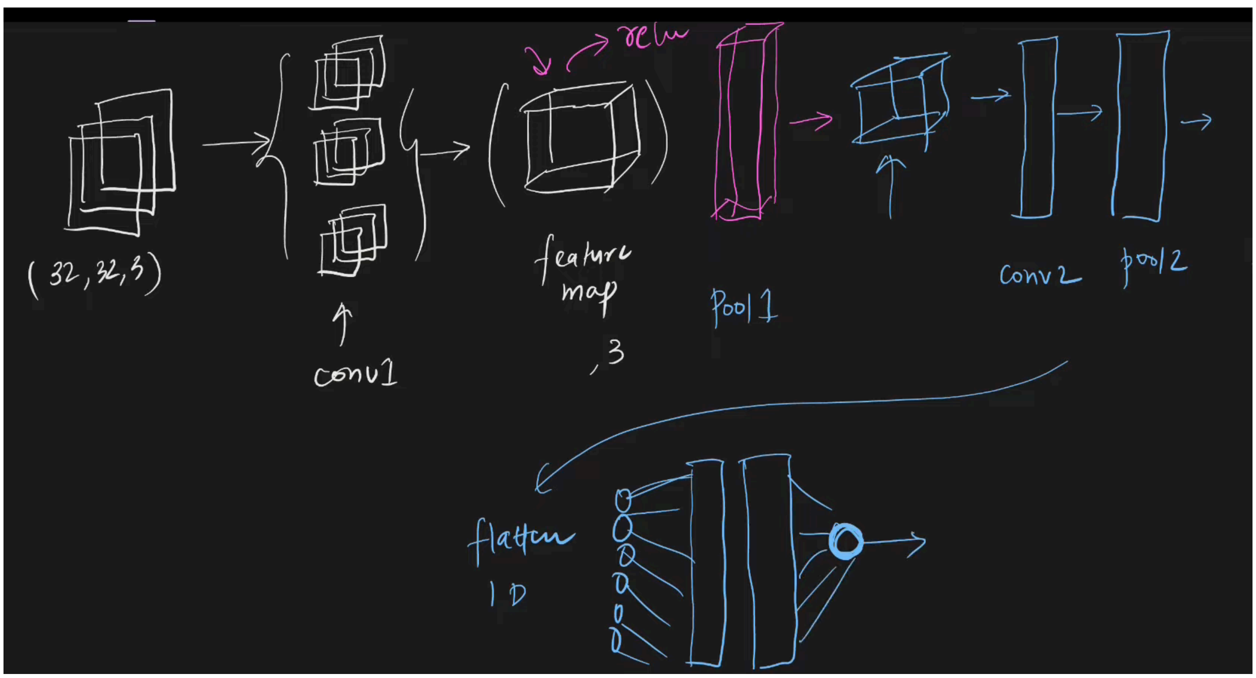
CNN Architecture



1. Input → RGB Image (eg. 32 × 32 ×3)
2. Filters (Conv1)
3. Feature Map
4. Apply activation function on feature map (eg. Relu)
5. Pass it through pooling layer
6. You'll get a **tensor**
7. **Conv2**
8. Repeat



9. **Flatten** the **tensor**
10. Pass it through 1 or more fully connected layers
11. Output layer



🔷
Input Layer → 🔶 **Convolution Layer** → 🔷 **Activation Function (ReLU)** →
🔶

Pooling Layer → (Repeat...) → 🔷 Flatten → 🔶 Dense (Fully Connected Layer) → 🔷 Output Layer

🔷 1. Input Layer

- Takes in the image as a tensor (e.g., 28×28 grayscale or 224×224×3 RGB).
- Just passes the image to the next layer.

```
Input: 28×28×1 (grayscale image)
```

🔶 2. Convolutional Layer (Conv2D)

🧱 *Extracts features like **edges, corners, patterns**.*

- Uses small filters/kernels (e.g., 3×3 or 5×5).
- Each filter slides (convolves) across the image and creates a **feature map**.

Key Parameters:

Parameter	Meaning	Default/Example
`filters`	Number of output feature maps	e.g., 32, 64
`kernel_size`	Size of filter window	(3, 3)
`strides`	Step size	(1, 1) (default)
`padding`	`'valid'` or `'same'`	`'valid'` (default)
`activation`	Usually ReLU	`'relu'`

💡 **In CNN, learnable parameters are not dependent of input.**

🔷 3. Activation Function (ReLU)

- ReLU = **Rectified Linear Unit**

- Converts negative values to 0 and keeps positive values.

```
f(x) = max(0, x)
```

✅ Helps network learn **non-linear features** (curves, shapes, etc.).

🔶 4. Pooling Layer

- Reduces the size of feature maps.
- Keeps only **important features**.

Types:

- **Max Pooling**: keeps max value
- **Average Pooling**: keeps average

Typical values:

```
pool_size = (2, 2), strides = 2
```

🔁 5. Repeat Convolution + Activation + Pooling

This is done **multiple times**, gradually **increasing depth** (more filters) and **decreasing size** (via pooling).

Example:

```
Conv (32 filters) → ReLU → Pool
Conv (64 filters) → ReLU → Pool
```

🔷 6. Flatten Layer

- Converts the 2D output of convolution into a 1D vector.

Example:

```
[3×3×128] → [1152]
```

This is needed before sending data to fully connected layers.

🔶 7. Fully Connected (Dense) Layers

- Traditional neural network layers.
- Used for final **decision-making or classification**.

Example:

```
Dense(128, activation='relu')
Dense(10, activation='softmax')  # 10 classes (digits 0-9)
```

🔚 8. Output Layer

- Produces the final prediction (e.g., class probabilities).
- Uses:
  - **Softmax** → for classification (multiclass)
  - **Sigmoid** → for binary classification

📊 Visual Structure of a CNN (Simplified)

```
Input Image (e.g. 28×28×1)
       ↓
[Conv2D] → [ReLU] → [Pooling]
       ↓
[Conv2D] → [ReLU] → [Pooling]
       ↓
[Flatten]
       ↓
```

```
[Dense Layer]
      ↓
[Output Layer (Softmax)]
```

Python Code

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

model = Sequential()

# Step 1: Convolution
model.add(Conv2D(32, kernel_size=(3,3), activation='relu', input_shape=(28,28,1)))

# Step 2: Pooling
model.add(MaxPooling2D(pool_size=(2,2)))

# Step 3: Second convolution layer
model.add(Conv2D(64, kernel_size=(3,3), activation='relu'))

# Step 4: Pooling again
model.add(MaxPooling2D(pool_size=(2,2)))

# Step 5: Flatten
model.add(Flatten())

# Step 6: Dense layer (fully connected)
model.add(Dense(128, activation='relu'))

# Step 7: Output layer (10 digits)
model.add(Dense(10, activation='softmax'))
```

```
model.summary()
```

Why is `input_shape=(28, 28, 1)`?

📌 It means:

- `28` → image **height** (number of rows)
- `28` → image **width** (number of columns)
- `1` → **channels** (color depth):
  - `1` = grayscale image (just shades of gray)
  - `3` = RGB color image (Red, Green, Blue)

📷 Example:

The **MNIST digit dataset** has:

- 28×28 grayscale images of handwritten digits.
- So each image is shaped:

  ```
  28 rows (height) × 28 columns (width) × 1 channel
  ```

📈 Typical Pattern for Number of Filters in Deep CNNs

Layer #	Filters	Reason
1st	32 or 64	Simple features like edges, blobs
2nd	64 or 128	More complex shapes, textures
3rd	128 or 256	Combinations of earlier features
4th+	256–512+	Very abstract, high-level features

🎯 How to Decide Number of Dense Layers & Neurons

✅ 1. Use 1 or 2 Dense Layers (Usually Enough)

Use Case	Example	Recommendation
Simple task (e.g., MNIST)	28×28 digits	1 Dense layer (e.g., 128)
Medium complexity (e.g., CIFAR-10)	RGB images	1–2 Dense layers (128 → 64)
High complexity (e.g., ImageNet)	Big models	2–3 Dense layers (1024 → 512 → 256)

✅ 2. Number of Neurons (Units) in Dense Layers

Factor	Strategy
After Flatten	Start with 128 or 256 units
Next Dense layer	Usually reduce (e.g., 128 → 64 → 32)
Output layer	Same as number of target classes

```
# Start with:
Dense(128)

# Then go down if needed:
Dense(64)
Dense(32)
```

🧪 Practical Example

✅ For MNIST (10-digit classification):

```python
Flatten()
Dense(128, activation='relu')
Dense(10, activation='softmax')  # 10 classes
```

✅ For CIFAR-10 (RGB images, 10 classes):

```python
Flatten()
Dense(256, activation='relu')
Dense(128, activation='relu')
Dense(10, activation='softmax')
```

❌ Disadvantages of CNN Architecture

Limitation	Why It Matters
Needs lots of labeled data	Training well needs thousands or millions of images
Computationally heavy	Needs powerful GPUs for deep architectures
Poor at spatial reasoning	Pooling removes exact locations (bad for segmentation)
Not ideal for non-grid data	Works only on image-like data (not text trees or graphs)
May not handle rotation well	CNNs learn shift-invariance but not full rotation invariance

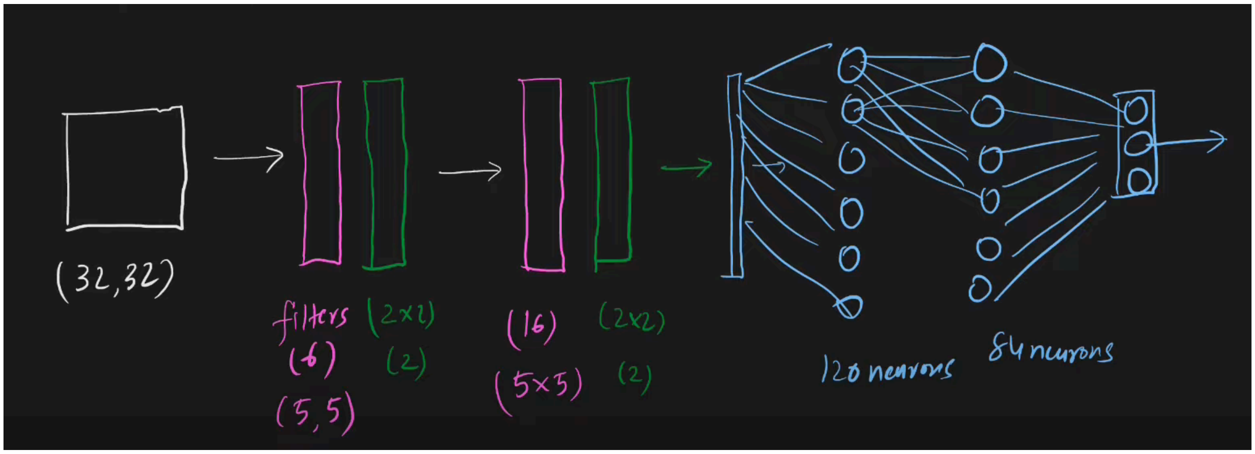
LeNet-5 (1998)

Application: Digit recognition (eg. MNIST)

5 → 5 Layers

Key Features:

- First successful CNN with stacked conv/pooling layers.
- Uses `tanh` activation (now replaced with ReLU).




```
model = Sequential()

model.add(Conv2D(6, kernel_size=(5,5), padding='valid', activation='tanh', input_shape=(32,32,1)))
model.add(AveragePooling2D(pool_size=(2, 2), strides=2, padding='valid'))

model.add(Conv2D(16, kernel_size=(5,5), padding='valid', activation='tanh'))
model.add(AveragePooling2D(pool_size=(2, 2), strides=2, padding='valid'))

model.add(Flatten())

model.add(Dense(120, activation='tanh'))
model.add(Dense(84, activation='tanh'))
model.add(Dense(10, activation='softmax'))
```

```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d_2 (Conv2D)           (None, 28, 28, 6)         156

 average_pooling2d_2 (Averag  (None, 14, 14, 6)        0
 ePooling2D)

 conv2d_3 (Conv2D)           (None, 10, 10, 16)        2416

 average_pooling2d_3 (Averag  (None, 5, 5, 16)         0
 ePooling2D)

 flatten_1 (Flatten)         (None, 400)               0

 dense_3 (Dense)             (None, 120)               48120

 dense_4 (Dense)             (None, 84)                10164

 dense_5 (Dense)             (None, 10)                850
```

Pros:

- Proved deep CNNs' effectiveness.
- Introduced ReLU and dropout.

Cons:

- High parameter count (~60M).
- Computationally expensive.