API Reference (Student-Friendly)
Lab 2: CAPTCHA Character Recognition with a CNN

Getting Started (Google Colab)

This lab is designed to run smoothly on **Google Colab**.

Upload these files into Colab:

- `lab_2_student_2026-01-23.ipynb`
- `lab_2_helpers.py`
- `captcha-images.tar.xz`

(*You are welcome to run this on your own computer using Anaconda Navigator or VS Code. In that case, you will need to set up the environment.*)

Extract the dataset inside Colab:

```
!tar -xf captcha-images.tar.xz
!ls
```

Important APIs You Will Use

This document explains **what each function does**, **why it matters**, and **common mistakes**.

Note: The following provides a guideline on how to implement the `[TODO]` tasks — for further references on specific function/library usage, please refer to the official documentation on the web, or you may use language models/ coding agents to ask about function usage and syntax.

OpenCV: Core Image Processing Functions

`cv2.imread(path)`

- **Purpose:** Load an image from disk into memory.
- **Output:** A NumPy array representing the image.
- **Common mistake:** If the path is wrong, OpenCV returns `None`.

`cv2.cvtColor(image, code)`

- **Purpose:** Convert between color spaces (e.g., BGR $\rightarrow$ grayscale).
- **Why needed:** This lab uses **grayscale** images for simplicity and consistency.

`cv2.copyMakeBorder(image, ...)`

- **Purpose:** Add padding around an image.
- **Why padding helps:** Characters near the border are less likely to get clipped during contour/cropping steps.

`cv2.threshold(gray, ...)`

- **Purpose:** Convert grayscale into black/white pixels (binary image).
- **Why needed:** Contour detection works best on binary images.

`cv2.findContours(binary, ...)`

- **Purpose:** Detect connected components (candidate character regions).
- **Tip:** External contours are typically enough for character segmentation.

`cv2.boundingRect(contour)`

- **Purpose:** Convert a contour into a bounding box (x, y, w, h).
- **Use:** Crop character images and sort them left-to-right.

Helper Utilities (`lab_2_helpers.py`)

`resize_to_fit(image, width, height)`

- **Purpose:** Resize a character image into a consistent shape (e.g., 20×20).
- **Why needed:** Neural networks require fixed input shapes.

`print_images(images, texts, n_rows, fig_size, ...)`

- **Purpose:** Display images as a grid with labels.
- **Why useful:** Helps verify if segmentation and labels are correct.

`unzip(list_of_tuples)`

- **Purpose:** Convert a list of pairs into two lists.
- **Example meaning:** `[(img1,"A"), (img2,"B")]` $\rightarrow$ `[img1,img2]` and `["A","B"]`.

Core Lab Functions (What They Do + Common Pitfalls)

`load_transform_image(image_path)`

Goal: Load one CAPTCHA image and convert it into a clean grayscale padded version.
What students should do (high-level):

1. Read the image from disk.
2. Convert it to grayscale.
3. Add padding around it (so cropping/segmentation is more reliable).

Common pitfalls:

- Not converting to grayscale ⇒ later steps behave unexpectedly.
- Using padding incorrectly ⇒ character shapes get distorted or clipped.

`extract_captcha_text(image_path)`

Goal: Extract the true label from the image filename. Example: `./captcha-images/2A2X.png` → `2A2X`
Key idea: The label is embedded in the filename (before `.png`).
Common pitfalls:

- Accidentally keeping `.png` in the text.
- Accidentally using the full path instead of the base filename.

`extract_chars(image)`

Goal: Extract exactly 4 character images from a CAPTCHA.
What students should understand:

1. Convert image into binary (foreground/background).
2. Find contours and convert them into bounding boxes.
3. Fix cases where a bounding box is "too wide" (two characters touching).
4. Sort boxes left-to-right and crop the 4 characters.

Common pitfalls:

- Getting fewer/more than 4 regions ⇒ discard that sample.
- Not sorting bounding boxes ⇒ characters appear in wrong order.

`make_feature(image)`

Goal: Convert a cropped character image into a CNN-ready input.
What students should do (high-level):

1. Resize the image to a fixed size (e.g., 20×20).
2. Ensure the shape includes a channel dimension: $(H, W) \rightarrow (H, W, 1)$.

Common pitfalls:

- Forgetting the channel dimension $(H, W, 1) \Rightarrow$ CNN shape mismatch.
- Returning inconsistent shapes $\Rightarrow$ training fails.

CNN Building Blocks (Beginner-Friendly)

This lab uses a **Convolutional Neural Network (CNN)**. A CNN is great for image recognition because it learns patterns like edges and curves automatically.

Output Dimension Formula (Conv2D)

For a 2D convolution with input width/height N, filter size F, stride S, and padding P:

$$N_{\text{out}} = \left\lfloor \frac{N - F + 2P}{S} \right\rfloor + 1$$

Special case: padding = "same"

- If padding is `"same"` and stride $S = 1$, then $N_{\text{out}} = N$.
- That means spatial size does not shrink due to convolution.

Output Dimension Formula (MaxPooling2D)

For a pooling layer with input size N, pooling window size F, stride S, and padding P:

$$N_{\text{out}} = \left\lfloor \frac{N - F + 2P}{S} \right\rfloor + 1$$

In most practical cases (including this lab), `MaxPooling2D` uses **no padding** ($P = 0$). For example, with $F = 2$ and $S = 2$, the spatial dimension is approximately halved.

Small CNN Example (10 Classes, Uses All Lab Layers)

This is a **toy example** (10 classes) that demonstrates the same **layer types** used in the lab:

- `Conv2D` (twice)
- `MaxPooling2D` (twice)
- `Flatten`
- `Dense` (hidden)
- `Dense + Softmax` (output)

Assumptions

- Input images are grayscale: $20 \times 20 \times 1$
- We want to classify into **10 classes**
- Conv uses:
  - filter size 5×5
  - stride $S = 1$
  - padding = `"same"` (so spatial dimensions stay the same)
- Pooling uses:
  - pool size 2×2
  - stride 2 (so spatial dimensions halve)

Dimension Walkthrough (Step-by-step)

Input: $20 \times 20 \times 1$

Conv1: 8 filters, 5×5, stride 1, `same` padding

$$20 \times 20 \times 1 \rightarrow 20 \times 20 \times 8$$

Pool1: 2×2, stride 2

$$20 \times 20 \times 8 \rightarrow 10 \times 10 \times 8$$

Conv2: 16 filters, 5×5, stride 1, `same` padding

$$10 \times 10 \times 8 \rightarrow 10 \times 10 \times 16$$

Pool2: 2×2, stride 2

$$10 \times 10 \times 16 \rightarrow 5 \times 5 \times 16$$

Flatten:

$$5 \times 5 \times 16 \rightarrow 5 \cdot 5 \cdot 16 = 400$$

Dense (hidden): 64 units

$$400 \rightarrow 64$$

Dense (output): 10 units + softmax

$$64 \rightarrow 10$$

Toy Code Snippet (For Understanding Only)

This snippet shows **how layers connect**. (**Your lab architecture may use different filter counts / dense sizes.**)

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

model = Sequential()

# Input: (20, 20, 1)
model.add(Conv2D(8, (5,5), strides=(1,1), padding="same",
                 activation="relu", input_shape=(20,20,1)))
# -> (20, 20, 8)

model.add(MaxPooling2D(pool_size=(2,2), strides=(2,2)))
# -> (10, 10, 8)

model.add(Conv2D(16, (5,5), strides=(1,1), padding="same",
                 activation="relu"))
# -> (10, 10, 16)

model.add(MaxPooling2D(pool_size=(2,2), strides=(2,2)))
# -> (5, 5, 16)

model.add(Flatten())
# -> (400,)

model.add(Dense(64, activation="relu"))
# -> (64,)

model.add(Dense(10, activation="softmax"))
# -> (10,)
```

Why these parameter choices?

- **Filter size 5×5:** captures slightly larger patterns than 3×3.
- **Stride 1 + same padding:** keeps spatial resolution stable inside each conv block.
- **Pooling 2×2:** reduces computation and creates robustness to small shifts.
- **Increasing filters (8 to 16):** later layers learn more complex patterns.
- **Softmax output:** converts the final layer into class probabilities.

Evaluation Logic (What It Should Track)

The lab compares predicted labels vs actual labels to compute how many predictions are correct, and to collect a few correct/incorrect examples for visualization.

Students should ensure:

- A counter increases only when prediction matches ground truth.
- Lists of indices store a few correct/incorrect sample positions (to display examples).
- The number of samples displayed is limited (to avoid printing too many).

Common pitfalls:

- Never updating counters $\Rightarrow$ accuracy displays as 0%.
- Never appending indices $\Rightarrow$ visualization shows nothing.