SI100B Project Face Detection and emotion classification

Lecture 3

Objectives

- Understanding the basic knowledge of convolutional neural networks.
- Get a preliminary understanding of the forward propagation.

Convolutional Neural Networks (CNNs) are a class of deep learning models designed primarily for processing structured grid data, such as images. They are highly effective for tasks like image recognition, object detection, and more, and have become a cornerstone of computer vision.

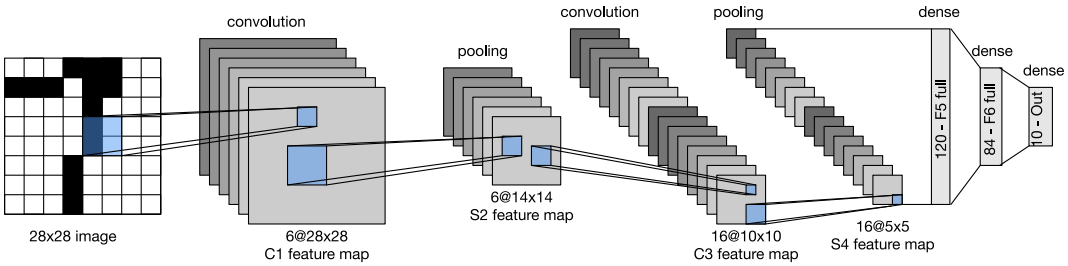

Figure 1: LeNet-5 architecture By Zhang, Aston and Lipton, Zachary C. and Li, Mu and Smola, Alexander J. - https://github.com/d2l-ai/d2l-en, CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=152265656

1 Key Components of CNNs

1.1 Convolutional Layers

- Perform convolution operations, which apply filters (kernels) to extract spatial features such as edges, textures, and patterns from the input image.
- Each filter slides over the image, creating a feature map that represents the presence of specific features.
- **Mathematical Definition:** The feature map output at position (i, j) is a sum of element-wise products between the input slice and the kernel:

$$\text{Feature\_map}(i,j) = \sum_{k=0}^{f_h-1} \sum_{l=0}^{f_w-1} \text{Input}(i+k, j+l) \cdot \text{Kernel}(k,l) + \text{Bias} \tag{1}$$

Put simply, image convolution means sliding a small "filter" (or kernel) over an image. At each position, we multiply the pixel values under the filter by the numbers in the kernel, add them up, and use that result as one pixel in the new image. It's like scanning the image and doing a small weighted sum at every step. This process helps the model extract local features from the image — things like edges, textures, or shapes. There are two hyper parameters in every convolution layer:

- **Stride**: Stride refers to the number of pixels by which the convolutional filter (or kernel) moves across the input image or feature map.

- **Padding**: Padding is the process of adding extra layers of zeros (or other values) around the border of the input before applying the convolution.

Output Size Formula: The output dimension can be calculated using the formula[cite: 41, 42, 43]:

$$\text{Output Size} = \lfloor \frac{\text{Input Size} - \text{Kernel Size} + 2 \cdot \text{Padding}}{\text{Stride}} \rfloor + 1$$

A typical Convolution function can be write as below:

```
def conv2d(x, kernel, stride=1, padding=0) -> np.array:
    pass
```

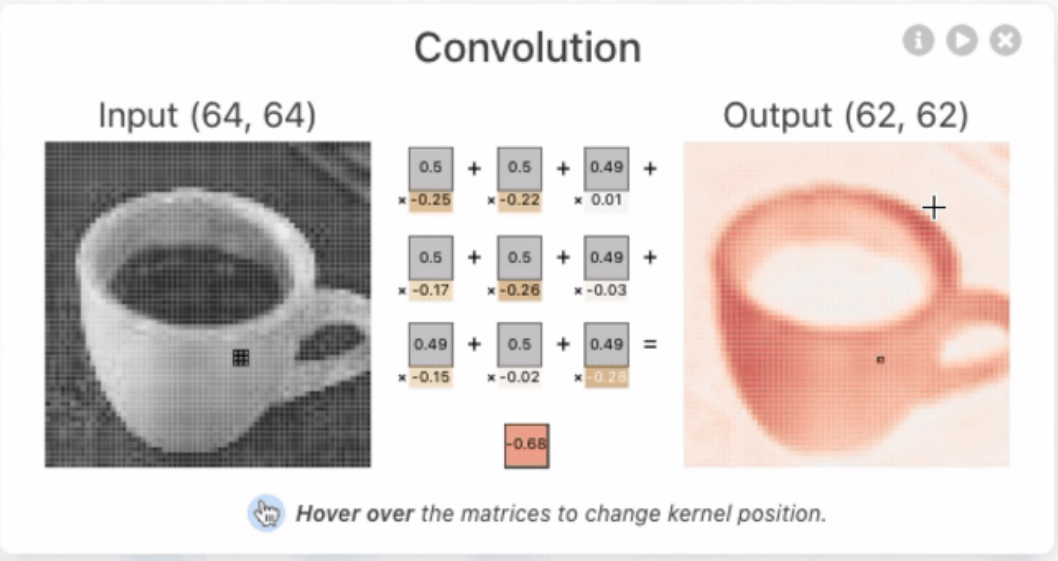

Figure 2: Convolutional opeator

1.2 Activation Function

- Introduced after each convolutional layer to introduce non-linearity, enabling the model to learn complex patterns.
- Common activation functions: ReLU (Rectified Linear Unit), Sigmoid, or Tanh.

$$\text{ReLU}(x) = \max(0, x), \quad \sigma(x) = \frac{1}{1 + e^{-x}}$$

A typical Sigmoid function can be write as below:

```
def Sigmoid(x) -> np.array:
    pass
```

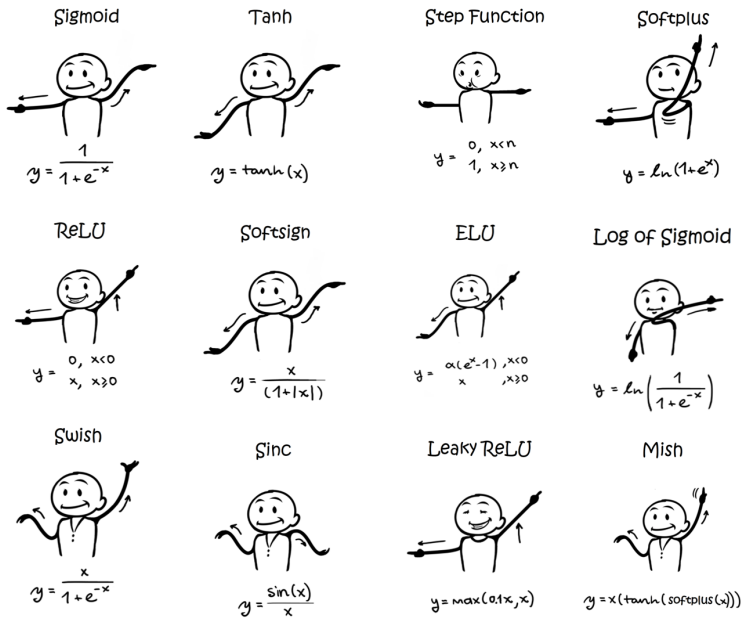

Figure 3: Activation function

1.3 Pooling Layers

Reduce the spatial dimensions (height and width) of feature maps, retaining essential features while minimizing computational cost. There are two types of pooling method.
Max Pooling:

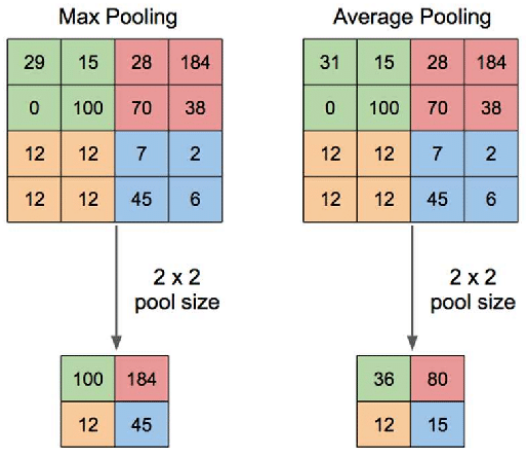

Figure 4: Pooling function

Pooling helps shrink the image while keeping the key features, which reduces parameters and helps prevent overfitting.
Output Size Formula: The output dimension can be calculated as:

$$\text{Output Size} = \lfloor \frac{\text{Input Size} - \text{Kernel Size}}{\text{Stride}} \rfloor + 1$$

1.4 Flattening

Flattening is the process of converting the multi-dimensional output of a convolutional layer (or layers) into a one-dimensional vector.

$$Output\_size = heigh \times width \times channels$$

1.5 Fully Connected Layers

Flatten the output of the convolutional and pooling layers into a single vector.
Serve as traditional neural network layers to perform final predictions, such as classification. It performs as :

$$y = xW + b$$

where W is the weight matrix and b is the bias vector. Please pay attention to matrix dimensions: in general x is treated as a row vector, but the vector formulas may differ slightly if it's handled differently.

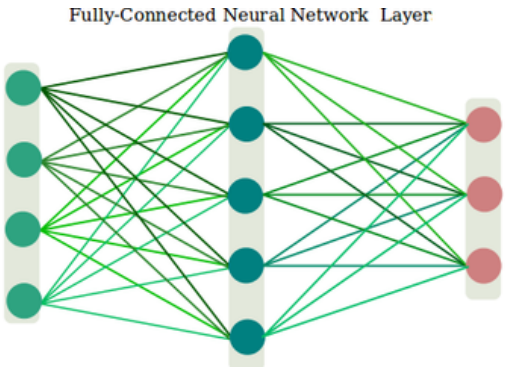

Figure 5: Pooling function

Visit this website to visualize CNN.

2 Task

Implement the following functions using **NumPy** only. Given a 48×48×1 image and a parameter matrix, the final output is obtained through one convolutional layer, followed by one pooling layer, one activation layer, and one fully connected layer.

```python
import numpy as np
import cv2

def conv2d(x, kernel, stride=1, padding=0) -> np.array:
    pass

def sigmod(X) -> np.array:
    pass

def max_pool_2d(x, pool_size=2, stride=2):
    pass
```

```python
def flatten(X) -> np.array:
    pass

def full_connect(X, W, b) -> np.array:
    pass

if __name__ == '__main__':
  #Read test case
  img = cv2.imread("sad.jpg", cv2.IMREAD_GRAYSCALE)
  #Normailize to [-1, 1)
  img = img / 127.0 - 1.0
  #Set parameters
  con_kernel = np.array([[0.05, 0.1, 0.05], [0.1, 0.4, 0.1], [0.05, 0.1, 0.05]])
  W = np.array([[0.01] * 529, [0.02] * 529, [0.03] * 529]).T
  b = np.array([1,1,1])

  # 1. Convolution
  out = conv2d(img, con_kernel)
  print(f"After Conv2D: {out.shape}")

  # 2. Activation (sigmod)
  out = sigmod(out)
  print(f"After sigmod: {out.shape}")

  # 3. Max Pooling
  out = max_pool_2d(out, pool_size=2, stride=2)
  print(f"After MaxPool2D: {out.shape}")

  # 4. Flatten
  out = flatten(out)
  print(f"After Flatten: {out.shape}")

  # 5. Fully Connected
  prob = full_connect(out, W, b)
  print(f"Final Output Scores: {prob}")
```

There is a bias in eq. 1. In our implementation, we omitted the bias term for simplicity.

3 Bonus

In deep learning, it is common during training to feed multiple images (a batch) into the model simultaneously. This is primarily done to improve computational efficiency, as modern hardware (such as GPUs) can process large amounts of data in parallel. This approach accelerates training and, through Mini-Batch Gradient Descent, yields gradient estimates that are more stable and efficient compared to using a single image at a time.
Upgrade to Multi-Batch and Multi-Channel Support. Modify the components you implemented to support the general input format used in practical deep learning frameworks.

1. **Input Data**: Should have the shape $(\text{Batch\_size}, H, W, \text{C}_{\text{in}})$.
2. **Convolution Kernels**: Should have the shape $(\text{Num\_filters}, \text{Kernel\_H}, \text{Kernel\_W}, \text{C}_{\text{in}})$.
3. **Convolution shape change**: $(\text{Batch\_size}, H, W, \text{C}_{\text{in}})$ -> $(\text{Batch\_size}, H, W, \text{Num\_filters})$

4. **Flatten shape change**: (Batch_size, H, W, C) -> (Batch_size, $H \times W \times C$)

Num_filters represent the number of output channels.

3.1 Test Case

For testing convenience, you will be provided with a convolutional weight matrix and a fully connected weight matrix. Apart from these two sets of weights, please set the biases of both the convolutional and fully connected layers to 0. The convolutional layer takes in a 3-channel image and outputs 10-channel data, while the fully connected layer ultimately produces 3 class labels. You will also be given 4 test images.
There are five steps in the test case:

1. Conv
2. ReLu Activation
3. MaxPool
4. flatten
5. fully connection

Form a batch using these four images and perform inference using the provided weight matrices—you will obtain a unique result. If your result matches mine exactly, you will receive full credit.

Please print the output list for easy verification