AI VIET NAM – COURSE 2022

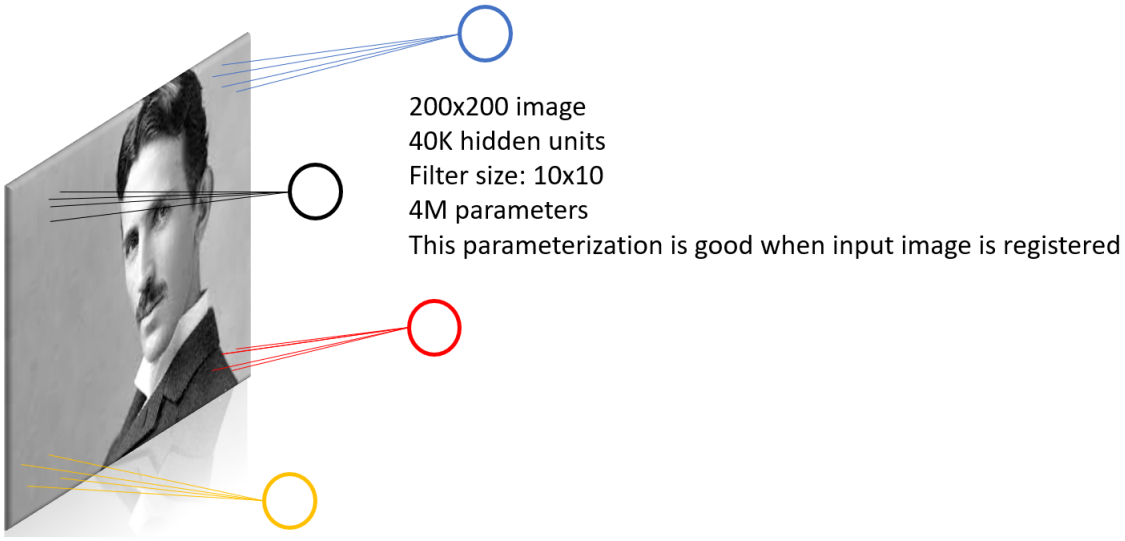
Basic CNN - Tutorial

Ngày 11 tháng 12 năm 2022

Phần I: Tổng quan Convolutional Neural Network (CNN)

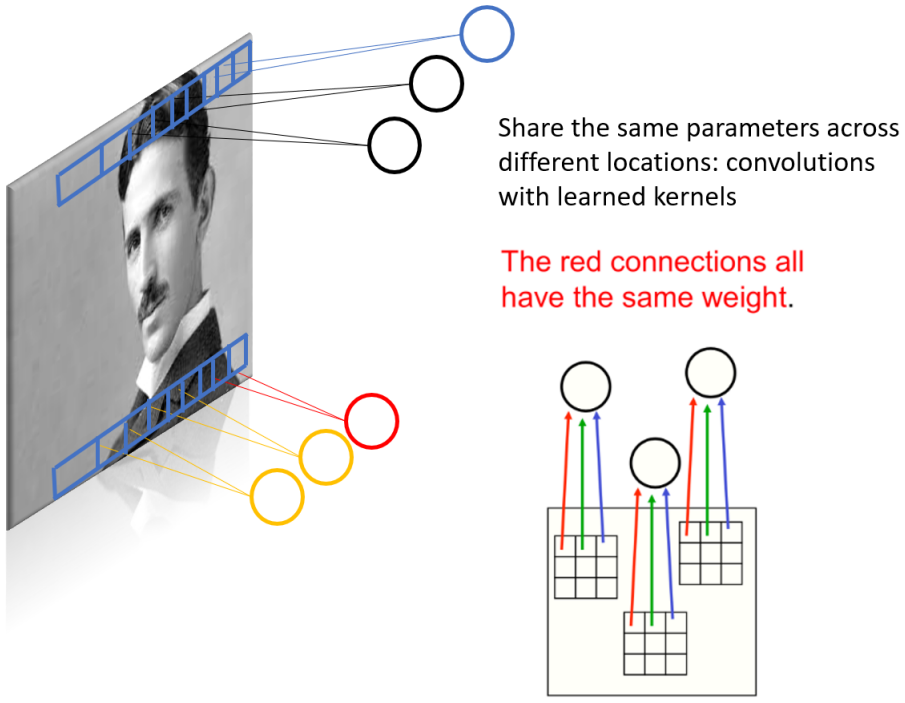
1. **Normal Neural Network trên images**
   Mỗi image có hàng ngàn pixels, mỗi pixel được xem như 1 feature, vậy nếu như một bức ảnh có kích thước 1000 * 1000, thì sẽ có 1.000.000 features. Với normal feed-forward neural networks, mỗi layer là full-connected với previous input layer.
   Trong noral feed-forward neural network, với mỗi layer, có 1.000.000 pixels, mỗi pixel lại kết nối full-connected với 1.000.000 pixels ở layer trước, tức sẽ có 1012 tham số. Đây là còn số quá lớn để có thể tính được vào thời điểm đó, bởi vì với mô hình có nhiều tham số, sẽ dễ bị overfitted và cần lượng lớn data cho việc training, ngoài ra còn cần nhiều memory và năng lực tính toán cho việc training và prediction.

2. **CNN hoạt động ra sao?**
   Có 2 đặc tính của image hình thành nên cách hoạt động của CNN trên image, đó là **feature localization** và **feature independence of location**.
   Feature localization: mỗi pixel hoặc feature có liên quan với các pixel quanh nó.
   Feature Independence of location: mỗi feature dù nó có nằm ở đâu trong bức ảnh, thì nó vẫn mang giá trị của feature đó. CNN xử lý vấn đề có quá nhiều tham số với Shared parameters (feature independence of location) của Locally connected networks (feature loclization), được gọi là Convolution Net.
   Locally connected layer: Trong hidden layer đầu tiên, mỗi node sẽ kết nối tới một cụm nhỏ pixels của input image chứ không phải toàn bộ image, gọi là small portion. Theo cách này, ta sẽ có ít kết nối hơn, vì thế ít tham số hơn giữa input và hidden layer đầu tiên.



Hình 1: Local connected layer

Shared parameters: Có nhưng khu vực, mà việc tìm ra feature là giống nhau về cách làm, vì vậy ta có thể dùng chung bộ parameter, trong hình trên là phía phải bên trên và phía trái bên dưới. Tức ta chia sẻ bộ parameter giữa những vị trí khác nhau trong bức ảnh.



Hình 2: Shared parameters

3. **Thành phần chính của CNN** Có 3 thành phần chính của CNN: convolution, pooling và fully-connected layer.
   Convulution layer: gồm filter và Stride. Filter là locally connected network, mỗi filter (kernel) sẽ học được nhiều feature trong images. Mỗi filter sẽ di chuyển quanh bức ảnh với bước nhảy được cấu hình trước là Stride.

Pooling layer: bao gồm max-pooling và average-pooling layer tương ứng. Max-pooling layer chọn giá trị lớn nhất từ cửa sổ tính toán, còn Average-pooling layer sẽ tính giá trị trung bình của mỗi window.
Fully-connected layer: flatten convolution layer cuối cùng và fully connect neuron cho output layer.

4. **Dataset - MNIST**
Để đơn giản, chúng ta sẽ dùng bộ dataset MNIST, về phân lớp chữ biết tay, yêu cầu là cho một image, phân loại nó thuộc lớp nào trong 10 ký tự số.

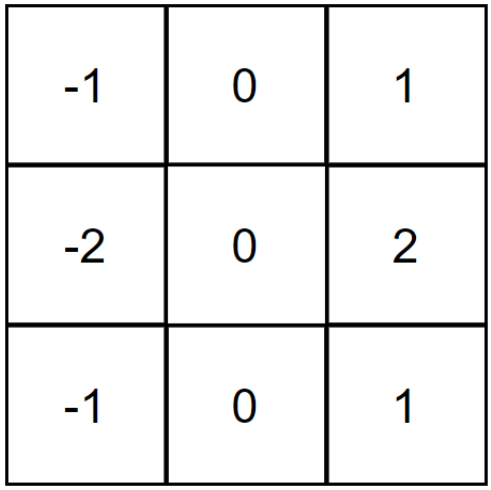
Hình 3: Dataset MNIST

Mỗi image trong MNIST dataset có 28x28 ký tự dạng grayscale, được phân bố tại trung tâm bức ảnh.
Với một neural network thông thường, đã có thể xử lý tốt bài toán này. Bằng cách chuyển image 28x28 thành vector có 28x28 = 784 chiều, và đưa vector này vào input layer, đi qua một vài hidden layers, và kết thúc bởi output layer với 10 nodes, mỗi node đại diện cho 1 ký tự số.
Lưu ý rằng, với MNIST dataset chứa các image nhỏ, và ký tự số nằm ở trung tâm bức ảnh, chứ nếu ký tự nằm ở các góc cạnh của bức ảnh, sẽ gặp khó khăn. Điều này là khó khăn với những bức ảnh trong thực tế khi cần phân loại.

5. **Các khái niệm quan trọng**

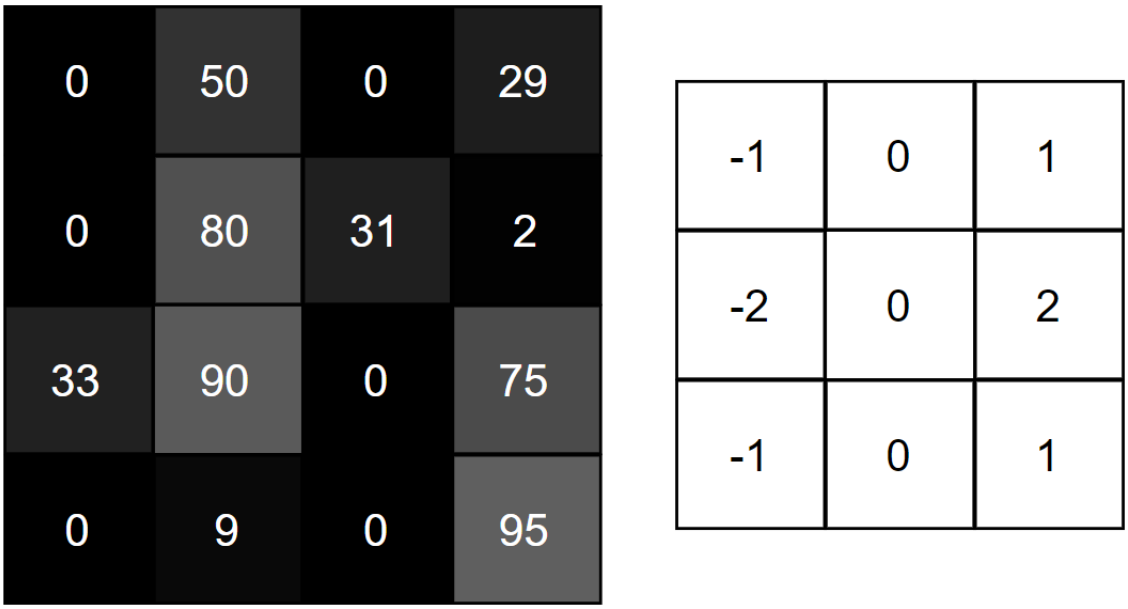
   (a) **Convolutions**
   Convolutional neural networks cơ bản chỉ là convolutional layers, gọi tắt là conv layers, hoạt động dựa trên phép toán convolution của toán học. Các conv layers bao gồm các filters (hay còn gọi là convolutional kernel, convolutional filter, hoặc kernel), nó đơn giản là matix 2 chiều gồm các con số. Ví dụ 3x3 filter.
   Chúng ta có thể dùng 1 imput image và 1 filter để tính output image bởi tích chập (convolving) filter với input image. Bao gồm các bước sau:

   - Đặt filter lên phía trên image tại vài vị trí.
   - Thực hiện element-wise multiplication giữa các giá trị trong filter và giá trị tương ứng trên image.
   - Cộng các element-wise products lại. Kết quả tổng sẽ là output value cho destination pixel trên output image.
   - Lặp lại cho các vị trí khác.



Hình 4: 3x3 filter

Có thể mô phỏng 4 bước trên bằng một ví dụ nhỏ, với bức ảnh 4x4 grayscale và 3x3 filter:
Các con số ở trên biểu diễn giá trị của pixel, trong khoảng từ 0 (black) đến 255 (white). Kết

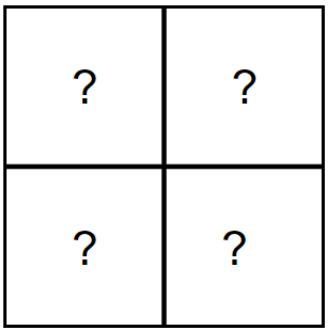


Hình 5: 4x4image-3x3filter

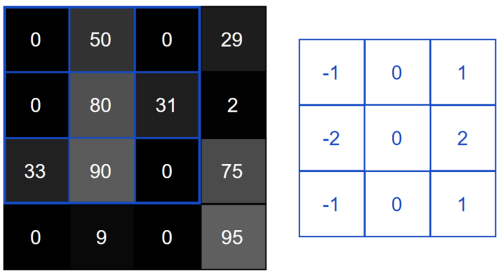
quả của phép tính input image và filter là matrix 2x2 có dạng như sau.
Đầu tiên ta đặt filter lên góc trái phía trên của input image:
Kết quả thu được là:
Tiếp theo cộng các kết quả lại: 62 – 33 = 29, và đặt nó vào output image.

Trường hợp là ảnh màu, thì input là tensor 3 chiều, vì vậy filter là tensor 3 chiều trượt hết trong khối input.

Nếu ta muốn detect multiple features, ví dụ như muốn tìm cả horizontal edges và vertical



Hình 6: cnn-2x2output-image

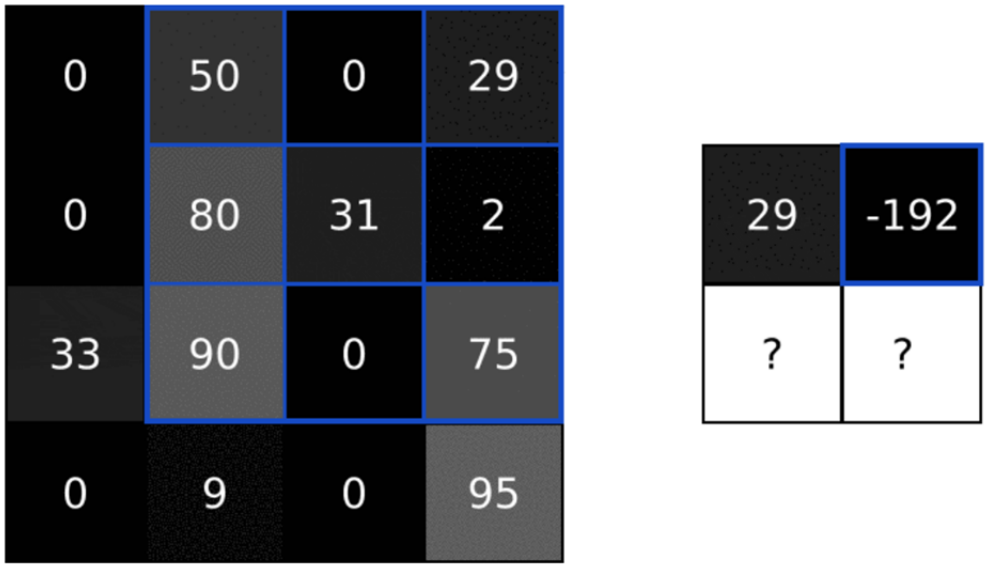


Hình 7: cnn-input-output

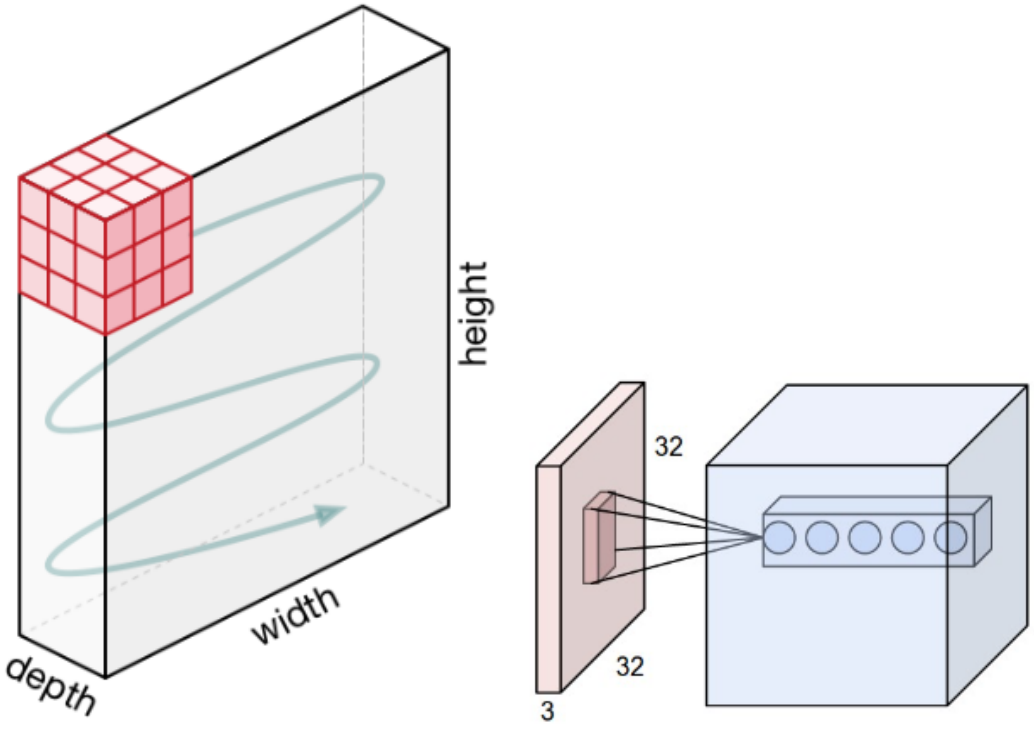
Image Value	Filter Value	Result
0	-1	0
50	0	0
0	1	0
0	-2	0
80	0	0
31	2	62
33	-1	-33
90	0	0
0	1	0

Hình 8: Bảng tính giá trị cho 1 pixel trên output image

edges, hoặc bất kỳ feature nào, ta sẽ muốn học từ nhiều convolutional filters trên cùng imput. Đó là ta muốn tính toán đồng thời cho các kernel khác nhau.



Hình 9: Output image



(a) convolution with tensor (b) Detect multiple features

Hình 10: Convolution with tensor and multiple feature

```
class Conv3x3:
  # A Convolution layer using 3x3 filters.

  def __init__(self, num_filters):
    self.num_filters = num_filters

```

```
    # filters is a 3d array with dimensions (num_filters, 3, 3)
    # We divide by 9 to reduce the variance of our initial values
    self.filters = np.random.randn(num_filters, 3, 3) / 9

  def iterate_regions(self, image):
    # Generates all possible 3x3 image regions using valid padding. Image is
    a 2d numpy array.

    h, w = image.shape

    for i in range(h - 2):
      for j in range(w - 2):
        im_region = image[i:(i + 3), j:(j + 3)]
        yield im_region, i, j

  def forward(self, input):
    # Performs a forward pass of the conv layer using the given input.
    # Returns a 3d numpy array with dimensions (h, w, num_filters). Input is
    a 2d numpy array

    self.last_input = input

    h, w = input.shape
    output = np.zeros((h - 2, w - 2, self.num_filters))

    for im_region, i, j in self.iterate_regions(input):
      output[i, j] = np.sum(im_region * self.filters, axis=(1, 2))

    return output

  def backprop(self, d_L_d_out, learn_rate):
    # Performs a backward pass of the conv layer. d_L_d_out is the loss
    gradient for this layer's outputs, learn_rate is a float.

    d_L_d_filters = np.zeros(self.filters.shape)

    for im_region, i, j in self.iterate_regions(self.last_input):
      for f in range(self.num_filters):
        d_L_d_filters[f] += d_L_d_out[i, j, f] * im_region

    # Update filters
    self.filters -= learn_rate * d_L_d_filters

    # We aren't returning anything here since we use Conv3x3 as the first
    layer in our CNN.
    # Otherwise, we'd need to return the loss gradient for this layer's
    inputs, just like every
    # other layer in our CNN.
    return None

```

(b) **Filter in computer vision**

Để ý rằng, filter 3x3 ở trên, trong thị giác máy tính, gọi là vertical Sobel filter, và ta có thêm một dạng khác là horizontal Sobel filter.

Khi áp dụng mỗi loại filter trên vào input image, output image sẽ có dạng:

Sobel filters là edge-detectors, vertical Sobel filter sẽ tìm vertical edges, và horizontal Sobel filter tìm horizontal edges. Với những pixel sáng hơn, tức có giá trị cao hơn (gần 255) ở output image. Vậy Sobel filter cho ta cách filter cạnh theo chiều dọc và ngang của input

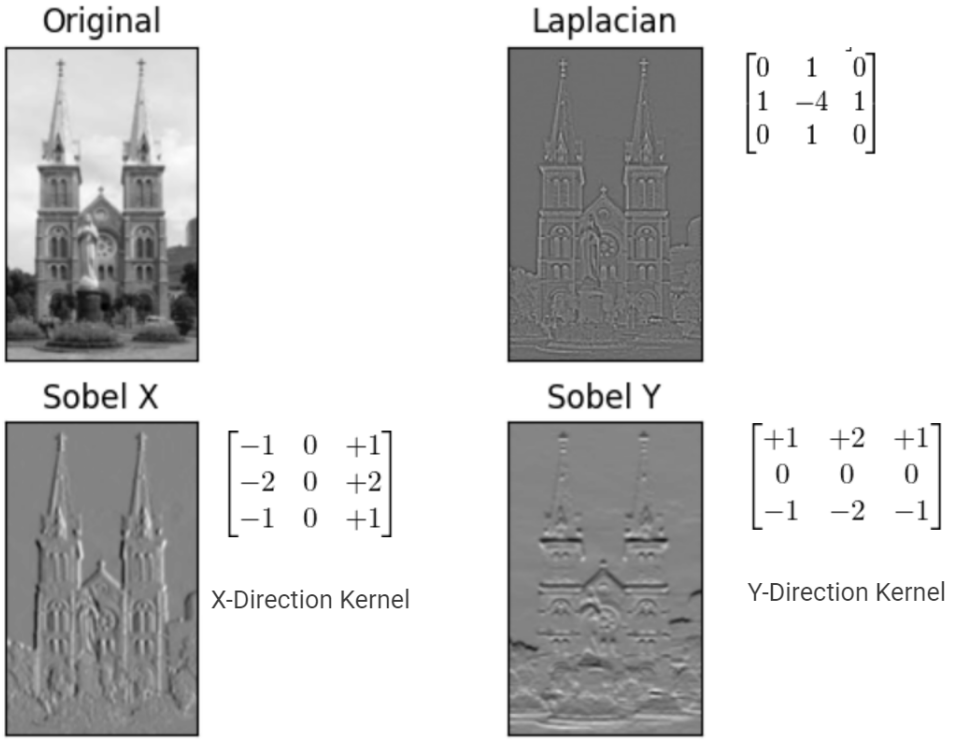
-1	0	1
-2	0	2
-1	0	1

(a) Vertical Sobel filter

1	2	1
0	0	0
-1	-2	-1

(b) Horizontal Sobel filter

Hình 11: Sobel filters



Hình 12: Sobel filter detect edges

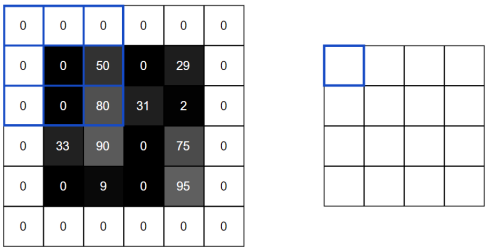
image.

Tại sao edge-detected image mang thông tin hữu ích hơn ảnh gốc?

Khi CNN train với dataset MNIST, dựa vào edge-detection filter và so sánh các vertical edges gần trung tâm bức ảnh sẽ dễ nhận diện thuộc ký tự nào hơn. Nói chung, convolution giúp ta tìm đặc trưng của ảnh tại các vị trí cụ thể (ví dụ edges).

(c) **Padding**

Khi tích chập 4x4 input image với 3x3 filter được 2x2 output image, nhưng nếu muốn giữ nguyên kích thước output image như input image, ta phải thêm các giá trị vào ngoài đường bao của input image, gọi là padding. Giá trị thêm vào có thể là zero hoặc 255 hoặc một giá trị trung bình nào đó. Với 3x3 filter sẽ cần 1 pixel padding, gọi là **same padding** (sẽ có dạng khác gọi là valid padding):

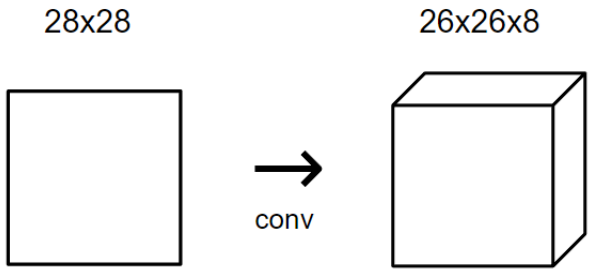


Hình 13: Same padding

(d) **Conv layer**

Từ các mục ở trên, ta đã biết image convolution hoạt động như thế nào và tại sao nó hữu dụng, tiếp theo sẽ tìm hiểu nó thực sự được dùng trong CNN ra sao. CNN bao gồm conv layer dùng các filter để xử lý các input images và cho ra output images. Tham số đầu tiên của conv layer là số lượng filter.

Trong MNIST CNN, chúng ta dùng một conv layer với 8 lớp filter như layer khởi tạo trong network, có nghĩa là 28x28 input image sẽ có 26x26x8 output volume (khối):



Hình 14: Conv layer

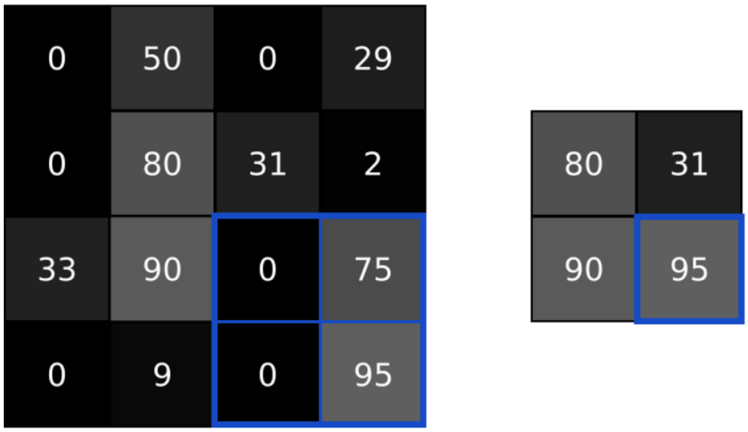
Ở đây output là 26x26x8 chứ không phải 28x28x8 vì ta dùng valid padding, nó sẽ giảm 2 pixels ở chiều ngang và chiều dọc. Ta có 3x3 (filter size) x 8 (số filters) = 72 tham số.

(e) **Pooling**

Các pixels kế nhau trong 1 bức ảnh, có giá trị gần giống nhau, vì vậy conv layer sẽ cũng tạo ra các giá trị giống nhau cho các pixels kế cạnh trong output. Vì vậy thông tin chứa trong các output của conv layer là gần giống nhau, nên ta sẽ không có được thông tin mới.

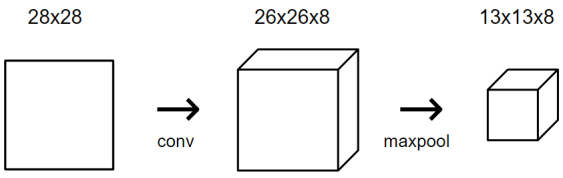
Pooling layer giải quyết được vấn đề này. Chúng giảm kích thước của input thông qua pooling values. Pooling thông thường được tính bằng min, max và average. Bên dưới là ví dụ về việc tính Max pooling với pooling size bằng 2:

Max pooling (pooling size = 2) trên 4x4 image sẽ giảm chiều output còn 2x2. Vậy mục tiêu của pooling là để thay đổi số lượng chiều dữ liệu, bằng cách chia chiều dọc và chiều ngang



Hình 15: Max Pooling with size 2x2

của input cho pool size.



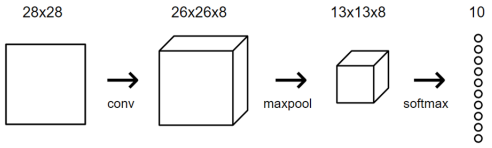
Hình 16: Max pooling reduce dimentions of tensor

```python
class MaxPool2:
  # A Max Pooling layer using a pool size of 2.

  def iterate_regions(self, image):
    # Generates non-overlapping 2x2 image regions to pool over, image is a 2d
     numpy array

    h, w, _ = image.shape
    new_h = h // 2
    new_w = w // 2

    for i in range(new_h):
      for j in range(new_w):
        im_region = image[(i * 2):(i * 2 + 2), (j * 2):(j * 2 + 2)]
        yield im_region, i, j

  def forward(self, input):
    # Performs a forward pass of the maxpool layer using the given input.
    # Returns a 3d numpy array with dimensions (h / 2, w / 2, num_filters),
    input is a 3d numpy array with dimensions (h, w, num_filters)

    self.last_input = input

    h, w, num_filters = input.shape
    output = np.zeros((h // 2, w // 2, num_filters))

    for im_region, i, j in self.iterate_regions(input):
      output[i, j] = np.amax(im_region, axis=(0, 1))
```

```
    return output

  def backprop(self, d_L_d_out):
    # Performs a backward pass of the maxpool layer.
    # Returns the loss gradient for this layer's inputs, d_L_d_out is the
    loss gradient for this layer's outputs.

    d_L_d_input = np.zeros(self.last_input.shape)

    for im_region, i, j in self.iterate_regions(self.last_input):
      h, w, f = im_region.shape
      amax = np.amax(im_region, axis=(0, 1))

      for i2 in range(h):
        for j2 in range(w):
          for f2 in range(f):
            # If this pixel was the max value, copy the gradient to it.
            if im_region[i2, j2, f2] == amax[f2]:
              d_L_d_input[i * 2 + i2, j * 2 + j2, f2] = d_L_d_out[i, j, f2]

    return d_L_d_input
```

(f) **Softmax**

Để hoàn tất CNN, ta phải đưa ra khả năng dự đoán, dùng standard final layer cho multiclass classification: softmax layer, một fully-connected (dense) layer dùng softmax function như là activation.



Hình 17: Softmax

Ở một khía cạnh khác, có thể chuyển output thành xác suất, vậy số nào có xác suất dự đoán cao thì nó mang ký số đó, ta thực sự không cần dùng softmax để dự đoán các con số, mà chỉ đi tìm những số có output cao nhất từ network.

```
class Softmax:
  # A standard fully-connected layer with softmax activation.

  def __init__(self, input_len, nodes):
    # We divide by input_len to reduce the variance of our initial values
    self.weights = np.random.randn(input_len, nodes) / input_len
    self.biases = np.zeros(nodes)

  def forward(self, input):
    # Performs a forward pass of the softmax layer using the given input.
    # Returns a 1d numpy array containing the respective probability values,
    input can be any array with any dimensions.

    self.last_input_shape = input.shape

```

```
    input = input.flatten()
    self.last_input = input

    input_len, nodes = self.weights.shape

    totals = np.dot(input, self.weights) + self.biases
    self.last_totals = totals

    exp = np.exp(totals)
    return exp / np.sum(exp, axis=0)

  def backprop(self, d_L_d_out, learn_rate):
    # Performs a backward pass of the softmax layer.
    # Returns the loss gradient for this layer's inputs. d_L_d_out is the
    loss gradient for this layer's outputs, learn_rate is a float
    # We know only 1 element of d_L_d_out will be nonzero
    for i, gradient in enumerate(d_L_d_out):
      if gradient == 0:
        continue

      # e^totals
      t_exp = np.exp(self.last_totals)

      # Sum of all e^totals
      S = np.sum(t_exp)

      # Gradients of out[i] against totals
      d_out_d_t = -t_exp[i] * t_exp / (S ** 2)
      d_out_d_t[i] = t_exp[i] * (S - t_exp[i]) / (S ** 2)

      # Gradients of totals against weights/biases/input
      d_t_d_w = self.last_input
      d_t_d_b = 1
      d_t_d_inputs = self.weights

      # Gradients of loss against totals
      d_L_d_t = gradient * d_out_d_t

      # Gradients of loss against weights/biases/input
      d_L_d_w = d_t_d_w[np.newaxis].T @ d_L_d_t[np.newaxis]
      d_L_d_b = d_L_d_t * d_t_d_b
      d_L_d_inputs = d_t_d_inputs @ d_L_d_t

      # Update weights / biases
      self.weights -= learn_rate * d_L_d_w
      self.biases -= learn_rate * d_L_d_b

      return d_L_d_inputs.reshape(self.last_input_shape)

```

Phần II: Triển khai CNN bằng Python và Keras

Ta đi thực hiện CNN với dataset MNIST, dùng 3 lớp: Conv 3x3, MaxPool và Softmax.

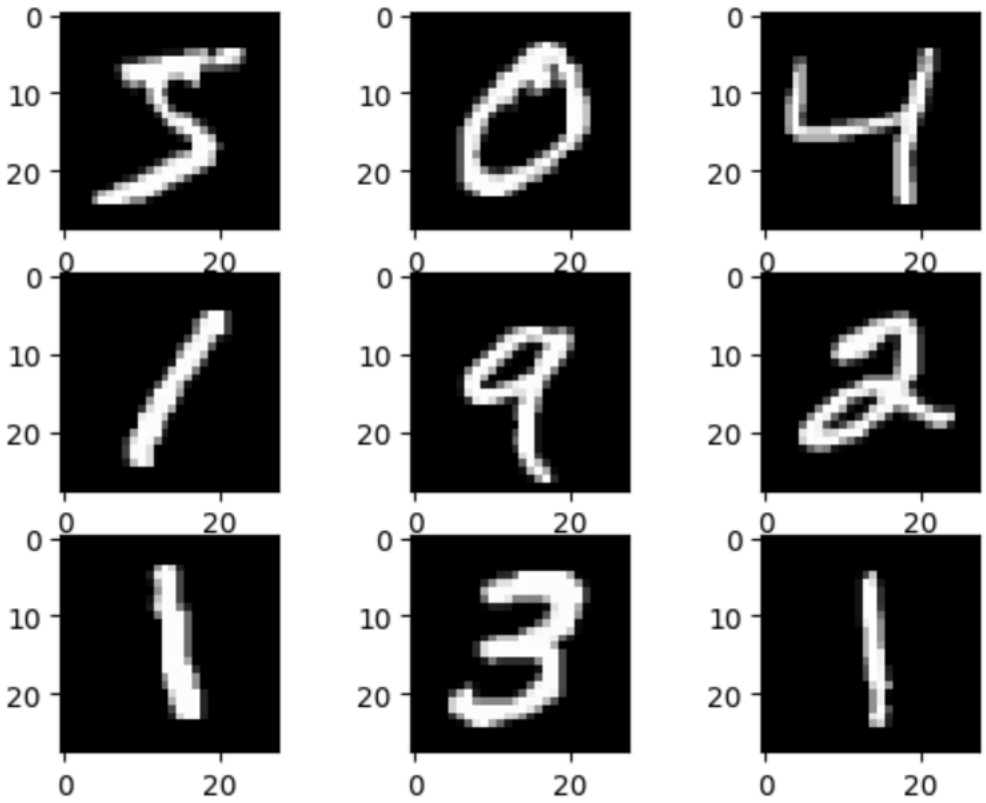
1. **import libraries và load dataset**

```
from keras.datasets import mnist
```

```
from matplotlib import pyplot
import numpy as np

#loading the dataset
(train_X, train_y), (test_X, test_y) = mnist.load_data()

# summarize loaded dataset
print('Train: X=%s, y=%s' % (train_X.shape, train_y.shape))
print('Test: X=%s, y=%s' % (test_X.shape, test_y.shape))
# plot first few images
for i in range(9):
  # define subplot
  pyplot.subplot(330 + 1 + i)
  # plot raw pixel data
  pyplot.imshow(train_X[i], cmap=pyplot.get_cmap('gray'))
# show the figure
pyplot.show()

```

Output:

```
Train: X=(60000, 28, 28), y=(60000,)
Test: X=(10000, 28, 28), y=(10000,)
```



Hình 18: cnn-implement-dataset

2. **CNN with keras**

```
from keras.models import Sequential
from keras.layers import Conv2D, MaxPooling2D, Dense, Flatten
from keras.utils import to_categorical
from keras.optimizers import SGD

learning_rate = 0.001
num_epochs = 30
batch_size = 1

train_images = train_X
train_labels = train_y
test_images = test_X
test_labels = test_y

train_images = (train_images / 255) - 0.5
test_images = (test_images / 255) - 0.5

train_images = np.expand_dims(train_images, axis=3)
test_images = np.expand_dims(test_images, axis=3)

model = Sequential([
  Conv2D(8, 3, input_shape=(28, 28, 1), use_bias=False),
  MaxPooling2D(pool_size=2),
  Flatten(),
  Dense(10, activation='softmax'),
])

model.compile(SGD(learning_rate=learning_rate), loss='categorical_crossentropy',
    metrics=['accuracy'])

history = model.fit(
  train_images,
  to_categorical(train_labels),
  batch_size=batch_size,
  epochs=num_epochs,
  validation_data=(test_images, to_categorical(test_labels)),
)

```

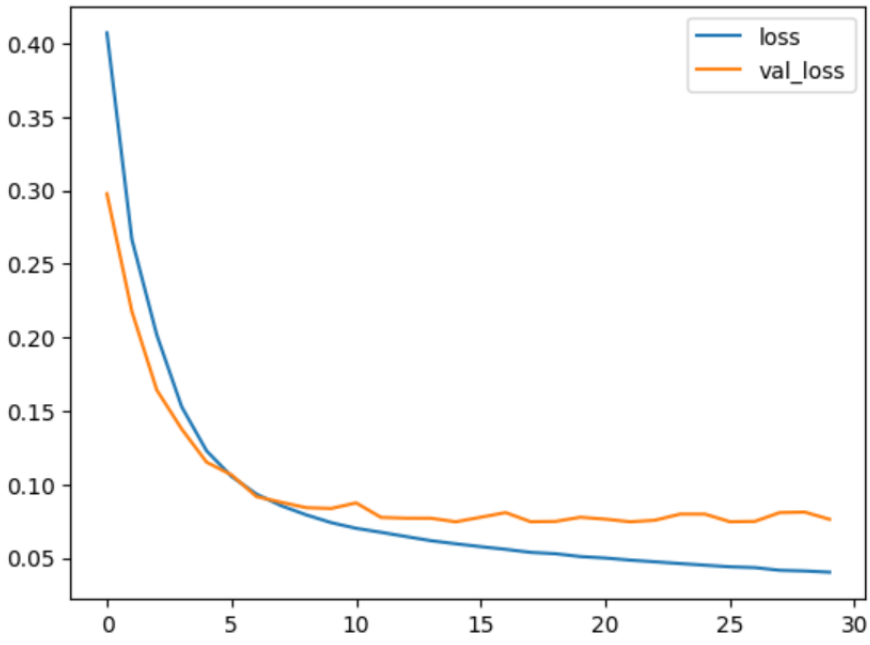
Output:

```
#plot the loss and validation loss of the dataset
pyplot.plot(history.history['loss'], label='loss')
pyplot.plot(history.history['val_loss'], label='val_loss')
pyplot.legend()

```

Output:

```
Epoch 7/30
60000/60000 [==============================] - 98s 2ms/step - loss: 0.0933 - accuracy: 0.9738 - val_loss: 0.0917 - v
Epoch 8/30
60000/60000 [==============================] - 102s 2ms/step - loss: 0.0854 - accuracy: 0.9758 - val_loss: 0.0877 -
Epoch 9/30
60000/60000 [==============================] - 88s 1ms/step - loss: 0.0792 - accuracy: 0.9768 - val_loss: 0.0841 - v
Epoch 10/30
60000/60000 [==============================] - 82s 1ms/step - loss: 0.0739 - accuracy: 0.9786 - val_loss: 0.0836 - v
Epoch 11/30
60000/60000 [==============================] - 66s 1ms/step - loss: 0.0701 - accuracy: 0.9795 - val_loss: 0.0874 - v
Epoch 12/30
60000/60000 [==============================] - 85s 1ms/step - loss: 0.0673 - accuracy: 0.9808 - val_loss: 0.0775 - v
Epoch 13/30
...
Epoch 29/30
60000/60000 [==============================] - 113s 2ms/step - loss: 0.0410 - accuracy: 0.9878 - val_loss: 0.0811 -
Epoch 30/30
60000/60000 [==============================] - 113s 2ms/step - loss: 0.0402 - accuracy: 0.9881 - val_loss: 0.0762 -
```

Hình 19: Kết quả train model dùng keras



Hình 20: So sánh loss giữa train và validation