Introduction to Convolutional Neural Networks (CNN) with TensorFlow

Learn the foundations of convolutional neural networks for computer vision and build a CNN with TensorFlow

Marco Peixeiro Follow
Apr 3 · 7 min read ★

Photo by Stephanie Cook on Unsplash

Recent advances in deep learning have made computer vision applications leap forward: from unlocking our mobile phone with our face, to safer self-driving cars.

Convolutional neural networks (CNN) are the architecture behind computer vision applications. In this post, you will learn about the foundations of CNNs and computer vision such as the convolution operation, padding, strided convolutions and pooling layers. Then, we will use TensorFlow to build a CNN for image recognition.

Spiderman recognizing Spiderman...

Understanding convolution

The **convolution** operation is the building block of a convolutional neural network as the name suggests it.

Now, in the field of computer vision, an image can be expressed as a matrix of RGB values. This concept was actually introduced in an earlier post.

To complete the convolution operation, we need an image and a filter.

Therefore, let's consider the 6x6 matrix below as a part of an image:

10	10	10	0	0	0
10	10	10	0	0	0
10	10	10	0	0	0
10	10	10	0	0	0
10	10	10	0	0	0
10	10	10	0	0	0

6x6 matrix. Source

And the filter will be the following matrix:

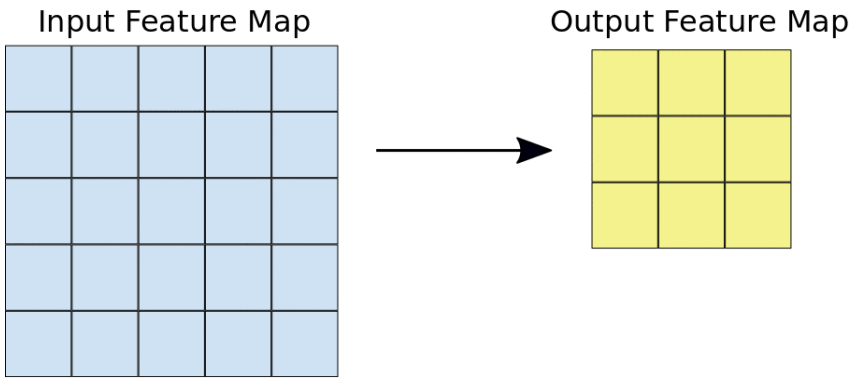
1	0	-1
1	0	-1
1	0	-1

3x3 filter. Source

Then, the convolution involves superimposing the filter onto the image matrix, adding the product of the values from the filter and and the

values from the image matrix, which will generate a 4x4 convoluted layer.

This is very hard to put in words, but here is a nice animation that explains the convolution:



Convolution operation

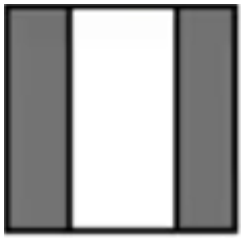
Performing this on the image matrix above and using the filter defined above, you should get the following resulting matrix:

0	30	30	0
0	30	30	0
0	30	30	0
0	30	30	0

4x4 output layer. Source

How do you interpret the output layer?

Well, considering that each value is indicative of color, or how dark a pixel is (positive value means light, negative value means dark), then you can interpret the output layer as:



Output layer interpretation. Source

Therefore, it seems that this particular filter is responsible to detect vertical edges in images!

How do you choose the right filter?

This is a natural question, as you might realize that there is an infinite number of possible filters you can apply to an image.

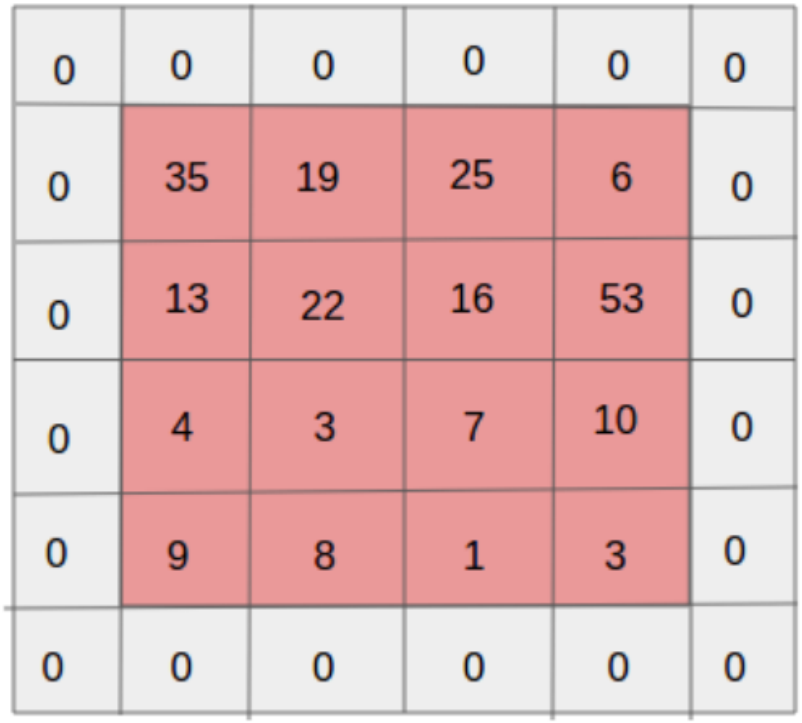
It turns out that the exact values in your filter matrix can be trainable parameters based on the model's objective. Therefore, you can either choose a filter that has worked for your specific application, or you can use backpropagation to determine the best values for your filter that will yield the best outcome.

Padding in computer vision

Previously, we have seen that a 3x3 filter convoluted with a 6x6 image, will result in 4x4 matrix. This is because there are 4x4 possible positions for the filter to fit in a 6x6 image.

Therefore, after each convolution step, the image shrinks, meaning that only a finite number of convolution can be performed until the image cannot be shrunken anymore. Furthermore, pixels situated in the corner of the image are only used once, and this results in loss of information for the neural network.

In order to solve both problems stated above, **padding** is used. Padding consists in adding a border around the input image as shown below:



Input image with padding of 1

As you can see, the added border is usually filled with zeros. Now, the corner pixel of the image will be used many times to calculate the output, effectively preventing loss of information. Also, it allows us to keep the input matrix shape in the output.

Considering our 6x6 input image, if we add a padding of 1, we get a 8x8 matrix. Applying a 3x3 filter, this will result in a 6x6 output.

A simple equation can help us figure out the shape of the output:

$$n + 2p - f + 1$$

Where n is the input shape, p is the padding size, and f is the filter shape

To reiterate, we have:

- 6x6 input

- padding of 1
- 3x3 filter

Thus, the output shape will be: 6+2(1)-3+1 = 6. Therefore, the output will be a 6x6 matrix, just like the input image!

Padding is not always required. However, when padding is used, it is usually for the output to have the same size as the input image. This yields two types of convolutions.

When no padding is applied, this is called a "**valid convolution**". Otherwise, it is termed "**same convolution**". To determine the padding size required to keep the dimensions of the input image, simply equate the formula above to n. After solving for p, you should get:

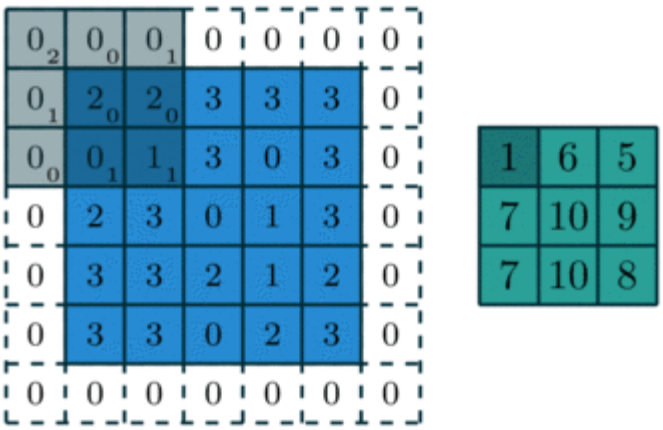
$$p = \frac{f - 1}{2}$$

You might have noticed that f should be odd in order for the padding to be a whole number. Hence, it is a convention in the field of computer visions to have odd filters.

Strided convolution

Previously, we have seen a convolution with a *stride* of 1. This means that the filter was moving horizontally and vertically by 1 pixel.

A strided convolution is when the stride is greater than 1. In the animation below, the stride is 2:



Convolution with a stride of 2

Now, taking into account the stride, the formula to calculate the shape of the output matrix is:

$$\frac{n + 2p - f}{s} + 1$$

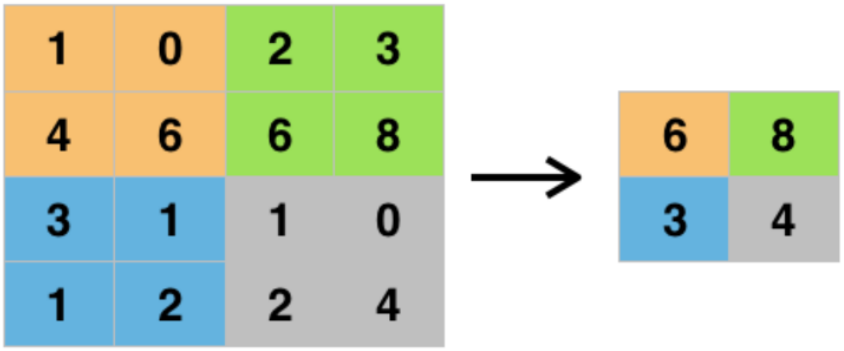
Where s is the stride

As a convention, if the formula above does not yield a whole number, then we round down to the nearest integer.

Pooling layers

Pooling layers are another way to reduce the size of the image interpretation in order to speed up computation, and it makes the detected features more robust.

Pooling is best explained with an image. Below is an example of **max pooling**:



Max pooling with a 2x2 filter

As you can see, we chose a 2x2 filter with a stride of 2. This is equivalent to dividing the input into 4 identical squares, we then take the maximum value of each square, and use it in the output.

Average pooling can also be performed, but it less popular than max pooling.

You can think of pooling as a way to prevent overfitting, since we are removing some features from the input image.

Why use a convolutional neural network?

We now have a strong foundational knowledge of convolutional neural networks. However, why do deep learning practitioners use them?

Unlike fully connected layers, convolutional layers have a much smaller set of parameters to learn. This is due to:

- parameter sharing
- sparsity of connection

Parameter sharing refers to the fact that one feature detector, such as vertical edges detector, will be useful in many parts of the image. Then, the sparsity of connections refers to the fact that only a few features are related to a certain output value.

Considering the above example of max pooling, the top left value for the output depends solely on the top left 2x2 square from the input image.

Therefore, we can train on smaller datasets and greatly reduce the number of parameters to learn, making CNNs a great tool for computer vision tasks.

Building a CNN with TensorFlow

Enough with the theory, let's code a CNN for hand signs recognition. We revisit a previous project to see if a CNN will perform better.

As always, the full notebook is available here.

Step 1: Preprocess the images

After importing the required libraries and assets, we load the data and preprocess the images:

```
# Loading the data (signs)
X_train_orig, Y_train_orig, X_test_orig, Y_test_orig, classes = load_dataset()

# Normalize images and use one-hot encoding
X_train = X_train_orig/255.
X_test = X_test_orig/255.
Y_train = convert_to_one_hot(Y_train_orig, 6).T
Y_test = convert_to_one_hot(Y_test_orig, 6).T

print ("number of training examples = " + str(X_train.shape[0]))
print ("number of test examples = " + str(X_test.shape[0]))
print ("X_train shape: " + str(X_train.shape))
print ("Y_train shape: " + str(Y_train.shape))
print ("X_test shape: " + str(X_test.shape))
print ("Y_test shape: " + str(Y_test.shape))
conv_layers = {}
```

Step 2: Create placeholders

Then, we create placeholders for the features and the target:

```
def create_placeholders(n_H0, n_W0, n_C0, n_y):

    X = tf.placeholder(tf.float32, [None, n_H0, n_W0, n_C0])
    Y = tf.placeholder(tf.float32, [None, n_y])

    return X, Y
```

Step 3: Initialize parameters

We then initialize our parameters using Xavier initialization:

```
def initialize_parameters():

    tf.set_random_seed(1)

    W1 = tf.get_variable("W1", [4, 4, 3, 8], initializer=tf.contrib.layers.xavier_initializer(seed=0))
    W2 = tf.get_variable("W2", [2, 2, 8, 16], initializer=tf.contrib.layers.xavier_initializer(seed=0))

    parameters = {"W1": W1,
                  "W2": W2}

    return parameters
```

Step 4: Define forward propagation

Now, we define the forward propagation step, which is really the architecture of our CNN. We will use a simply 3-layer network with 2 convolutional layers and a final fully-connected layer:

```python
def forward_propagation(X, parameters):

    # Retrieve the parameters from the dictionary "parameters"
    W1 = parameters['W1']
    W2 = parameters['W2']

    # CONV2D: stride of 1, padding 'SAME'
    Z1 = tf.nn.conv2d(X,W1,strides=[1, 1 ,1 ,1],padding='SAME')

    # RELU
    A1 = tf.nn.relu(Z1)

    # MAXPOOL: window 8x8, sride 8, padding 'SAME'
    P1 = tf.nn.max_pool(A1,ksize=[1,8,8,1],strides=[1,8,8,1],padding='SAME')

    # CONV2D: filters W2, stride 1, padding 'SAME'
    Z2 = tf.nn.conv2d(P1,W2,strides=[1,1,1,1],padding='SAME')

    # RELU
    A2 = tf.nn.relu(Z2)

    # MAXPOOL: window 4x4, stride 4, padding 'SAME'
    P2 = tf.nn.max_pool(A2,ksize=[1,4,4,1],strides=[1,4,4,1],padding='SAME')

    # FLATTEN
    P2 = tf.contrib.layers.flatten(P2)

    # FULLY-CONNECTED without non-linear activation function (not not call softmax).
    # 6 neurons in output layer. Hint: one of the arguments should be "activation_fn=None"
    Z3 = tf.contrib.layers.fully_connected(P2,6, activation_fn=None)

    return Z3
```

Step 5: Compute cost

Finally, we define a function that will compute the cost:

```python
def compute_cost(Z3, Y):

    cost = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits_v2(logits=Z3, labels=Y))

    return cost
```

Step 6: Combine all functions into a model

Now, we combine all the functions above into a single CNN network. We will use mini-batch gradient descent for training:

```
def model(X_train, Y_train, X_test, Y_test, learning_rate=0
          num_epochs=400, minibatch_size=64, print_cost=Tru

    ops.reset_default_graph()                         # to
    tf.set_random_seed(1)                             # to
    seed = 3                                          # to
    (m, n_H0, n_W0, n_C0) = X_train.shape
    n_y = Y_train.shape[1]
    costs = []                                        # To

    # Create Placeholders of the correct shape
    X, Y = create_placeholders(n_H0, n_W0, n_C0, n_y)

    # Initialize parameters
    parameters = initialize_parameters()

    # Forward propagation: Build the forward propagation in
    Z3 = forward_propagation(X, parameters)

    # Cost function: Add cost function to tensorflow graph
    cost = compute_cost(Z3, Y)

    # Backpropagation: Define the tensorflow optimizer. Use
    optimizer = tf.train.AdamOptimizer(learning_rate=learni

    # Initialize all the variables globally
    init = tf.global_variables_initializer()

    # Start the session to compute the tensorflow graph
    with tf.Session() as sess:

        # Run the initialization
        sess.run(init)

        # Do the training loop
        for epoch in range(num_epochs):

            minibatch_cost = 0.
            num_minibatches = int(m / minibatch_size) # num
            seed = seed + 1
            minibatches = random_mini_batches(X_train, Y_tr
```

```
42
43              for minibatch in minibatches:
44
45                  # Select a minibatch
46                  (minibatch_X, minibatch_Y) = minibatch
47
48                  # Run the session to execute the optimizer
49                  _ , temp_cost = sess.run([optimizer, cost],
50
51                  minibatch_cost += temp_cost / num_minibatch
```

Great! Now, we can run our model and see how it performs:

```
_, _, parameters = model(X_train, Y_train, X_test, Y_test)
```

In my case, I trained the CNN on a laptop using only CPU, and I got a pretty bad result. If you train the CNN on a desktop with a better CPU and GPU, you will surely get better results than I did.

. . .

Congratulations! You now have a very good knowledge about CNNs and the field of computer vision. Although there is much more to learn, the more advanced techniques use the concepts introduced here as building blocks.

In the next post, I will introduce residual networks with Keras!

Cheers!

Reference: deeplearning.ai