Convolutional neural networks

Use Python to implement a simple network that classifies handwritten digits

James Cowley

May 02, 2018

Learn about convolutional neural networks (CNN) and see how to use Python to implement a simple network that classifies handwritten digits.

At some point in your daily life, you've probably seen some form of an object recognition algorithm in action, for example, face detection on your phone's camera. How does this work, though? At the heart of computer vision solutions such as these are convolutional neural networks (CNN). Simply put, these are neural networks that are particularly adept at building complex features from less complex ones. A classic example is a face detector, where early layers pick out vertical and horizontal lines while later stages find noses and mouths.

This article explains how these convolutional nets work. It also shows how to use Python to implement a simple network that classifies handwritten digits. So, let's jump straight in!

Primer on neural networks

This article does not go into detail on how neural nets work in general, but you do need a little background before tackling convolutional nets. Neural networks have a layered architecture. Each layer comprises some number of nodes, and each node effectively conducts some mathematical operation on some input to calculate an output. The input to any given node is a weighted sum of the outputs of the previous layer (plus a bias term usually equal to one or zero). It is these weights that are learned by the algorithm during training. To learn these parameters, the output of a training run is compared with the true value, and the error is backpropagated through the network to update the weights.

Convolution

Convolution is a mathematical operation, where a function is "applied" in some manner to another function. The result can be understood as a "mixture" of the two functions. Convolution is represented by an asterisk (*), which might be confused with the * operator that is generally used for multiplication in many programming languages.
How does this help detect objects in an image, though? Well, it turns out that convolutions are really good at detecting simple structures in an image, and then putting those simple features together to construct even more complex features. In a convolutional network, this process occurs over a series of many layers, each of which conduct a convolution on the output of the previous layer.

So what sort of convolutions do you use in computer vision? To understand that, you must first understand exactly what an image is. An image is an array of bytes, either Rank 2 (two-dimensional, having a width and a height) or Rank 3 (three-dimensional, with width, height, and more than one channel). So a grayscale image is Rank 2, while an RGB image is Rank 3 (with three channels). The values of the bytes are simply interpreted as integer values, describing the amount of that particular channel that must be used on the corresponding pixel. So basically, when dealing with computer vision, you imagine an image as a 2D array of numbers (for an RGB or RGBA image, three or four such arrays overlaid on each other).

Therefore, my convolution takes this array (I am going to assume that the image is grayscale for now) and convolves it with a second array, called a filter. The convolution process proceeds as follows. First, the filter is overlaid in the upper left of the image array. Next, the elementwise product of the filter is taken with the subsection of the image over which the filter currently lies. That is, the upper-left element of the filter is multiplied by the upper-left element of the image, and so on. These results are then added to produce one value. The filter is then moved along the image by a distance that is called the stride, and the process is repeated. The output of this is a new array, of different dimensions than the image array (usually the result has a smaller width and height, but more channels). To illustrate how this works, look at an example. Here is a 3 x 3 filter:

\[
\begin{bmatrix}
-1 & 2 & -1 \\
-1 & 2 & -1 \\
-1 & 2 & -1 \\
\end{bmatrix}
\]

The following image is the one I'll apply this filter to.
After applying one pass of the filter to this image, I get the following result.
Hopefully, you can see that the filter seems to be arranged with the values going vertically. This means that it picks out vertical features in the image, as you can see in the result. It is the filter values that are learned when running a CNN.

It should be noted that the stride and filter size are hyperparameters, meaning that they are not learned by the model. So you must apply your scientific mind to work out what values of these quantities will work best for your application.

A final concept that you need to understand about convolution is the idea of padding. If your image won't fit the filter in an integer number of times (with the stride taken into account), then you must pad the image. There are two ways of doing this: VALID padding and SAME padding. The former basically drops any remaining values on the edge of the image. That is, if the filter is 2 x 2 with a stride of 2, and the image has a width of 3, then VALID padding ignores the third column of values from the image. Meanwhile, SAME padding adds values (usually zeros) to the edges of the images to increase its dimension until the filter can fit an integer number of times. This padding is generally done symmetrically (that is, it tries to add the same number of columns/rows on either side of the image).

It is also interesting to note that image convolutions have uses not limited just to computer vision. A lot of image filtering techniques can be implemented using convolution, for example, blurring and sharpening.
The following basic Python code shows how the convolution operation works (you could make this neater by using numpy, for example):

```python
def basic_conv(image, out, in_width, in_height, out_width, out_height, filter, filter_dim, stride):
    result_element = 0
    for res_y in range(out_height):
        for res_x in range(out_width):
            for filter_y in range(filter_dim):
                for filter_x in range(filter_dim):
                    image_y = res_y + filter_y
                    image_x = res_x + filter_x
                    result_element += (filter[filter_y][filter_x] * image[image_y][image_x])
            out[res_y][res_x] = result_element
            result_element = 0
            res_x += (stride - 1)
            res_y += (stride - 1)
    return out
```

Note that if you want to write the result out to an image file (to visualize it like I did above), then you must clamp the output values so that they do not exceed 255.

**Pooling and fully connected layers**

Real convolutional networks are rarely built from just convolutional layers. They usually have other types of layers, too. The simplest is the fully connected layer. This is just a normal neural network layer where all of the outputs of the previous layer are connected to all of the nodes on the next layer. Typically, these layers come toward the end of the network.

The other key type of layer you will see in convolutional nets is the pooling layer. This comes in a few forms, but the most commonly used is max pooling, in which the input matrix is split into equal-sized segments, and the maximum value in each segment is taken forward to fill the corresponding element of the output matrix.

$$
\begin{bmatrix}
17 & 9 & 3 & 5 \\
6 & 20 & 4 & 7 \\
9 & 0 & 61 & 5 \\
10 & 10 & 2 & 8 \\
\end{bmatrix}
\rightarrow
\begin{bmatrix}
20 & 7 \\
10 & 61 \\
\end{bmatrix}
$$

In the code listing above, the input was split into 2 x 2 quadrants, and max pooling was applied. Therefore, I can describe this particular operation as having a filter of dimension 2 and a stride of 2. What this process has done is picked out the broad sectors in which a feature lies. Imagine that this network is looking for faces. In this case, you can interpret the result of this pooling as showing that there is a strong possibility of there being a face in the lower right, some possibility of a face in the upper left, and that there is likely no face in the upper right or lower left.
def max_pool(input, out, in_width, in_height, out_width, out_height, kernel_dim, stride):
    max = 0
    for res_y in range(out_height):
        for res_x in range(out_width):
            for kernel_y in range(kernel_dim):
                for kernel_x in range(kernel_dim):
                    in_y = (res_y * stride) + kernel_y
                    in_x = (res_x * stride) + kernel_x
                    if input[in_y][in_x] > max:
                        max = input[in_y][in_x]
                out[res_y][res_x] = max
    max = 0
    return out

Example background

Now, let's work through a simple computer vision problem by making a network that identifies handwritten digits in the image. This is one of the most common baseline examples used to show off the power of neural nets. The example is written in Python, using the TensorFlow library, so that you do not have to focus too much on specific implementation details and can instead look more at the overall architecture. TensorFlow has another benefit. It provides the MNIST data set built-in, though it should be noted that other machine learning frameworks (such as SciKit-Learn) also do this.

For training and testing, I use this MNIST data set. I use a relatively simple convolutional network architecture based on LeNet-5. This achieved an error rate of 0.9% on the MNIST set, though I won't hit quite that level of accuracy because I'll be foregoing a lot of the manipulations that LeCun and others conducted to make the network perform better, and I will be simplifying certain aspects of the architecture.

The full code can be found at my GitHub repository. This repository also includes the code for demonstrating the earlier edge-detection image filter.

Architecture

The architecture that I will be using is as follows:

1. A convolutional layer, reducing the 32x32x1 MNIST image to a 28x28x6 output
2. A max-pooling layer, halving the width and height of the features
3. A convolutional layer, bringing the dimensions to 10x10x16
4. A max-pooling layer, again halving the width and height
5. A fully connected layer, bringing the number of features down from 400 to 120
6. A second fully connected layer
7. A final fully connected layer, outputting a vector of size 10

Each intermediate layer uses a ReLU nonlinearity and each convolutional layer uses a 5x5 filter with a stride of 1 and VALID padding. Meanwhile, the max-pooling filter has dimension 2.
Helper methods
The code has several helper methods that abstract away some of the details that are repeated in
the architecture, such as creating filters (each filter is 5x5, but has different depths) and
convolutional layers. Note that I use a truncated Gaussian distribution in the weights initialization
because it does not matter what the weights start as, so long as they are not all identical. This is to
break symmetry. The following code shows an example of one of the helper methods.

```python
def make_conv_layer(self, input, in_channels, out_channels):
    layer_weights = self.init_conv_weights(in_channels, out_channels)
    layer_bias = self.make_bias_term(out_channels)
    layer_activations = tf.nn.conv2d(input, layer_weights, strides =
        self.conv_strides, padding = self.conv_padding) + layer_bias
    return self.relu(layer_activations)
```

Constructing the network
With the specific details of creating the layers abstracted out, the network construction is relatively
simple.

```python
def run_network(self, x):
    # Layer 1: convolutional, ReLU nonlinearity, 32x32x1 --> 28x28x6
    c1 = self.make_conv_layer(x, 1, 6)
    # Layer 2: Max Pooling. 28x28x6 --> 14x14x6
    p2 = self.make_pool_layer(c1)
    # Layer 3. convolutional, ReLU nonlinearity, 14x14x6 --> 10x10x16
    c3 = self.make_conv_layer(p2, 6, 16)
    # Layer 4. Max Pooling. 10x10x16 --> 5x5x16
    p4 = self.make_pool_layer(c3)
    # Flattening the features to be fed into a fully connected layer
    fc5 = self.flatten_input(p4)
    # Layer 5. Fully connected. 400 --> 120
    fc5 = self.make_fc_layer(fc5, 400, 120)
    # Layer 6. Fully connected. 120 --> 84
    fc6 = self.make_fc_layer(fc5, 120, 84)
    # Layer 7. Fully connected. 84 --> 10. Output layer, so no ReLU.
    fc7 = self.make_fc_layer(fc6, 84, 10, True)
    return fc7
```

Training
First, I split the MNIST set into a training set, a cross-validation set, and a test set.

```python
x_train, y_train, x_valid, y_valid, x_test, y_test = split()
x_train = pad(x_train)
x_valid = pad(x_valid)
x_test = pad(x_test)
x_train_tensor = tf.placeholder(tf.float32, (None, 32, 32, 1))
y_train_tensor = tf.placeholder(tf.int32, (None))
y_train_one_hot = tf.one_hot(y_train_tensor, 10)
```
The training labels are reformed as a one-hot vector. A one-hot vector is a vector where each element represents a class, and the element is equal to one if the example belongs to that class, and is equal to zero otherwise. So for an image depicting the number 1, the one-hot representation would be:

\[
\begin{bmatrix}
0 \\
1 \\
0 \\
0 \\
0 \\
0 \\
0 \\
0 \\
0 \\
0 \\
\end{bmatrix}
\]

I then set up the operations for defining how I am going to train the model, for example, defining what quantity I want to minimize during the training.

```python
net = lenet.LeNet5()
logits = net.run_network(x_train_tensor)

learn_rate = 0.001
cross_ent = tf.nn.softmax_cross_entropy_with_logits(logits = logits, labels = y_train_one_hot)
loss = tf.reduce_mean(cross_ent) # We want to minimise the mean cross entropy
optimisation = tf.train.AdamOptimizer(learning_rate = learn_rate)
train_op = optimisation.minimize(loss)

correct = tf.equal(tf.argmax(logits, 1), tf.argmax(y_train_one_hot, 1))
accuracy = tf.reduce_mean(tf.cast(correct, tf.float32))
```

I use the cross-entropy as my measure of loss, and use the AdamOptimiser to conduct the optimization.

The training set is split into batches, of size 128, and I run the training over some number of epochs (ten in this case). After each epoch, the resulting set of weights is checked against the cross-validation set.
with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    example_count = len(x_train)
    for i in range(num_epochs):
        x_train, y_train = shuffle(x_train, y_train)
        for j in range(0, example_count, batch_size):
            batch_end = j + batch_size
            batch_x, batch_y = x_train[j : batch_end], y_train[j : batch_end]
            sess.run(train_op, feed_dict={x_train_tensor: batch_x,
                                           y_train_tensor: batch_y})
        accuracy_valid = eval(x_valid, y_valid)
        print("Accuracy: {}".format(accuracy_valid))
        print()
    save.save(sess, "SavedModel/Saved")

The resulting model is then saved out to be used later on the test set.

**Performance**

When run against the test set, the model obtained an accuracy of 98.5% (a 1.5% error rate). This is a slightly lower performance than LeCun's implementation due to differences in data preparation and so on. However, for a relatively simple implementation such as mine, this is quite a high performance.

**Conclusion**

In this article, you learned the basics of convolutional neural networks, including the convolution process itself, max pooling, and fully connected layers. You then looked at an implementation of a comparatively simple CNN architecture. Hopefully, this will help you understand CNNs when you use them in your work, or help start your journey into learning about this fascinating branch of machine learning.

© Copyright IBM Corporation 2018

**Trademarks**
(www.ibm.com/developerworks/ibm/trademarks/)