# Convolutional Neural Networks

ECE 532

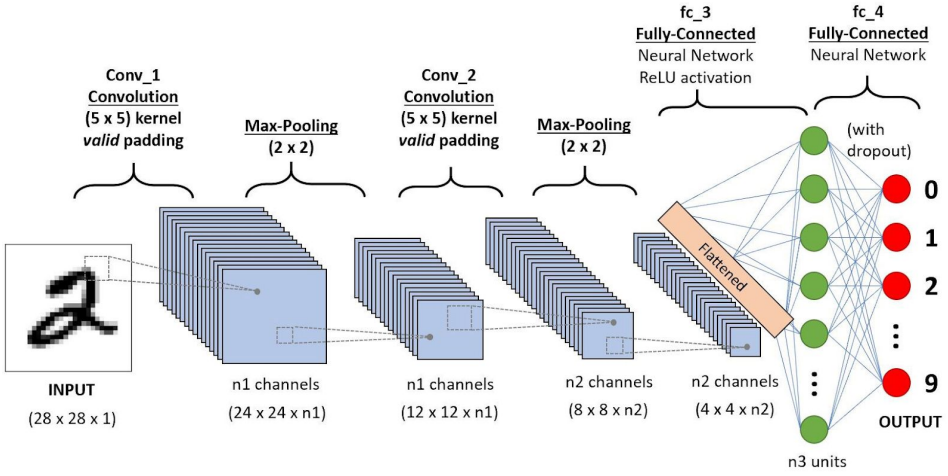
May 1st, 2020


| | | |
|---|---|---|
| Tessa McChesney | tmcchesney@wisc.edu | 907 319 2701 |
| Luis Guzman | ljguzman@wisc.edu | 906 982 9985 |
| Yong-Cheol Cho | ycho98@wisc.edu | 907 796 4402 |

## Abstract

Convolutional neural networks (CNN) have become the method of choice for solving many cutting-edge problems in computer vision and image recognition. They are a topic of ongoing research in engineering and computer science, and are at the core of many famous commercial artificial intelligence systems. They function by using multiple layers of convolution and pooling prior to a standard fully-connected neural network to allow the computer to learn a set of patterns to classify the data. In this project, we explain how these convolutional and pooling layers work. In the warm-up activities, students will be able to generalize how dimensions of inputs and outputs change between each CNN layer, as well as experiment with pooling layer code to visualize the results of pooling. The main activity is broken up into three parts. In the first part, students will complete three computational exercises corresponding to the math used in the three main layers of a CNN. Second, students will use and edit code provided to see how different filters affect an input image of Bucky at a convolutional layer. Lastly, students will see how the different layers fit together and how critical filters are to the success of the CNN, where we apply these methods to the problem of handwritten digit recognition. Though a true CNN could achieve much better results, the simple structure we chose to illustrate these concepts resulted in an 8% error rate when classifying the testing dataset. After completing these activities, we expect students to understand the benefits of a CNN, how convolutional and pooling layers work individually and together, and how it differs from a traditional neural network. CNNs that are used in practice have more complicated structures, but after successfully completing this lesson and exercises, students should have the basic understanding of these more complicated networks.

## Background




A typical structure of a CNN. Source: Elia, E. (2019)

In a traditional neural network, there is typically only one layer of input nodes. When applied to image recognition, the usual method would then be to unfold and stack the 2D image into a single column vector. This works well for simple tasks, but would fail for many modern computer vision problems. The

issue is that traditional neural networks are not designed to search for spatial patterns in the images, and instead rely entirely on global relationships between pixels for classification.

Convolutional neural networks aim to solve this issue by using a combination of convolution and pooling layers to extract localized patterns from the image prior to a traditional fully-connected neural network for classification. This idea was first proposed in a paper in 1998 by Yann LeCun et al., who was inspired by how the human visual cortex processes patterns from light. LeCun applied the concept to the MNIST dataset for handwritten digit recognition, and used the technology for a commercial system to verify the writing on bank checks. This work was gradually expanded upon and in 2012, Alex Krizhevsky et al. unveiled AlexNet, which could classify the 1.2 million image ImageNet dataset with a top-5 error rate of 17.0%. More recent CNNs can achieve an error rate of 3.6% on this dataset.

As with any machine learning technique, CNNs can have issues with selection bias in the training data and can run into ethical issues when used to classify people (for example, facial recognition software). When people of minority age, race, and gender groups are not adequately represented in the training data, many CNNs have had higher error rates when used by people of these groups. Current research, such as a 2017 paper by Howard et. al, is investigating the best methods for accounting for this bias. Regardless of the methods used to correct for it, avoiding selection bias is an essential part of ethical engineering practices when designing a CNN.

In the following sections, we will explain the concepts of convolution and pooling layers independently, and show how these techniques can lead to more accurate image classification.

**Convolution**

In the convolutional layer of a CNN, the operation can be thought of as moving a predefined filter across the image, and calculating a new value for each pixel. The equation for the convolution operation is

$$h[m,n] = \sum_{k,l} g[k,l]\, f[m-k, n-l]$$

where *f* is the original image, *g* is a matrix consisting of some filter coefficients, *h* is the resulting convolved matrix, and *k* and *l* are the dimensions of g. For each element in the output matrix, we simply compute the weighted sum of surrounding elements, as specified by the filter matrix. We will illustrate this operation with the following example:

Take an image of an outline of a square. Here “1” corresponds to a bright pixel, and “0” is a dark pixel. We want to blur this image, so an averaging filter is used[1].

[1] For simplicity, we omitted the factor of 1/9 that should be present in each element of the filter matrix.

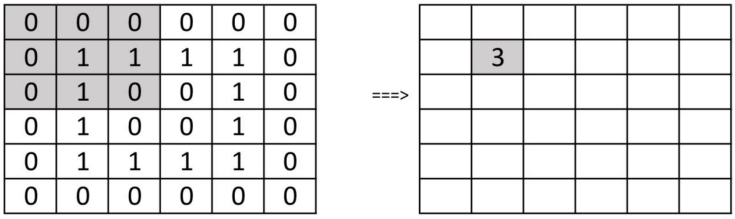
Image

| | | | | | |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 1 | 1 | 0 |
| 0 | 1 | 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 0 | 1 | 0 |
| 0 | 1 | 1 | 1 | 1 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 |

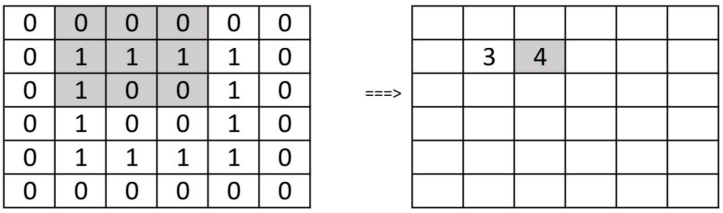
Filter

| | | |
|---|---|---|
| 1 | 1 | 1 |
| 1 | 1 | 1 |
| 1 | 1 | 1 |

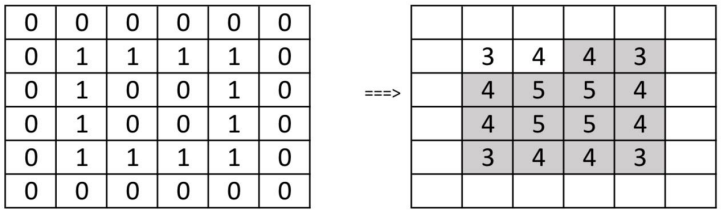
To compute the first element, the filter is overlaid in the top-left of the image. We then compute the sum of all 9 elements covered by the filter, where each element is multiplied by the filter coefficient covering them.



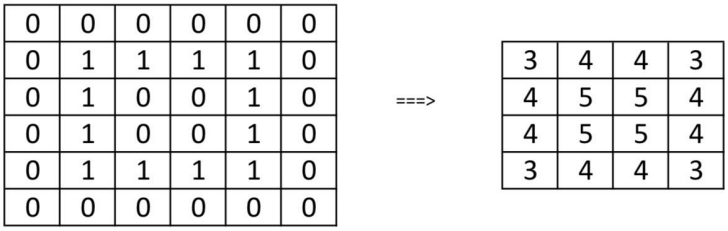

The next element is computed in the same fashion.



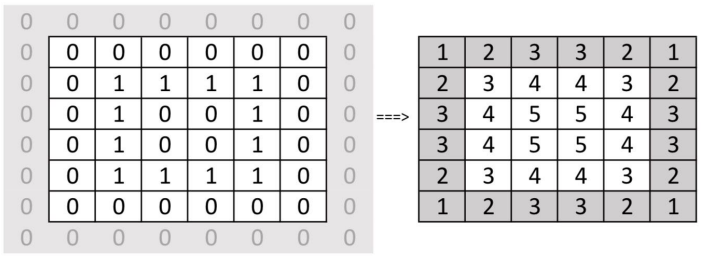

Similarly, we can fill the rest of the interior.




Now, what about the edges? Moving the filter to the edge would cause the filter to be outside dimensions of our original image. This brings up the concept of *padding*. We then have two options: crop the resulting matrix to reflect only those positions that are valid or we can produce a result matrix that is the same size as our image by expanding the image by a row of zeros on every side. These methods are called *valid padding* and *same padding* respectively.




Valid padding (Dimensions: 6x6 ==> 4x4)




Same padding (Dimensions: 6x6 ==> 6x6)

The last step in the convolution layer is to apply the rectified linear (ReLU) function to eliminate any negative values in our convolved matrix. This function sets all negative values to zero while keeping the positive values unchanged. It should be noted here that the ReLU function is non-linear, and these non-linearities play an important role in allowing our neural networks to learn arbitrary functions.

The ReLU function. $f(x) = x$ for $x > 0$, $f(x) = 0$ for $x < 0$

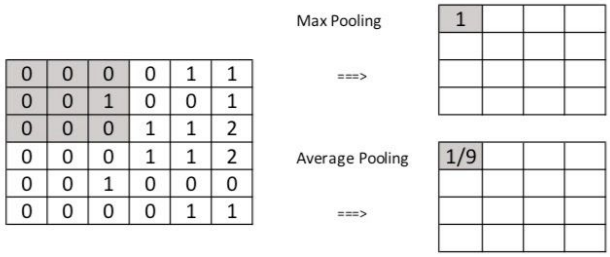
Figure created by authors using Desmos

Through this relatively simple convolution operation, we can produce filters that can extract complex patterns from our images. Oftentimes in a CNN, multiple convolutional layers are used. The first layer usually performs simple operations like edge detection or low-pass filtering. The deeper layers can build on these extracted features to form more complex features like an entire limb of an animal or wheel of a car. In order for these deeper layers to extract these features, we must reduce the dimensionality of our problem. Since a typical image can be over a million pixels in size, our goal is to only keep the information and features that are important to classifying the images. The next topic, pooling, will provide an essential method of dimension reduction.

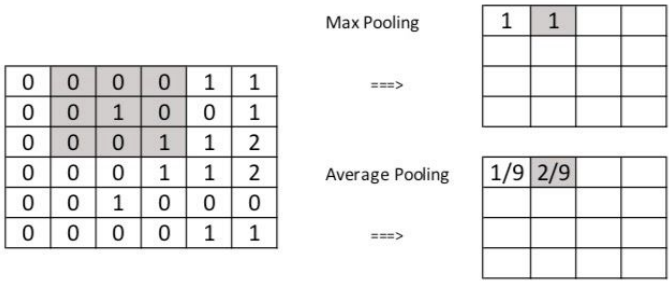
**Pooling**

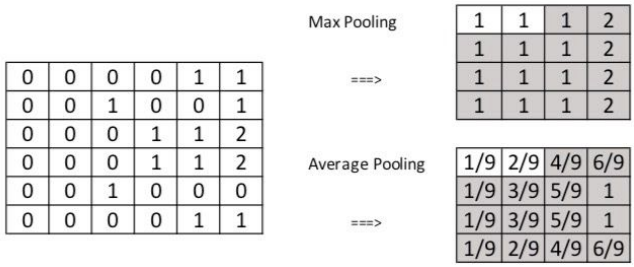
The goal of pooling layers is to reduce the amount of data to be processed by sub-sampling the input data. There are two main methods of pooling: *max pooling* and *average pooling*. Both methods work by moving a kernel across the input matrix and calculating a single output value for multiple inputs. Max pooling will simply take the maximum value and average pooling takes the average. In general, max pooling performs better since it also functions as a filter to block unwanted noise in the data.

In the following example, we start with an image of our square after it has been sent through an edge detector and ReLU function. The 3x3 pooling kernel starts by calculating the top-left value, just like in convolution.

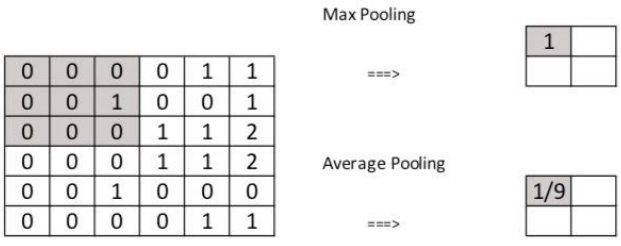



Similarly for the rest of the values:

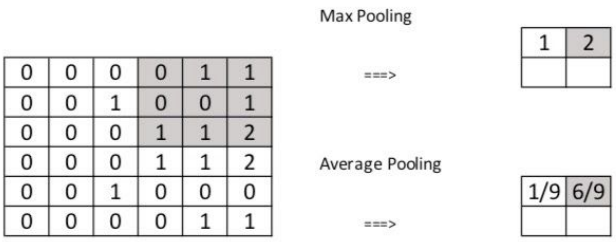


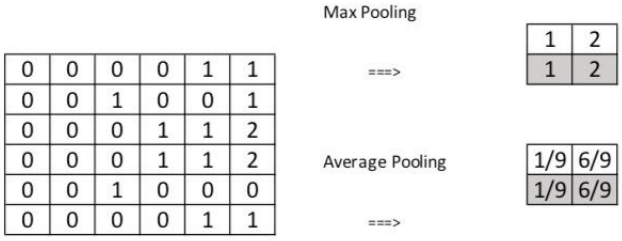

Both convolution and pooling layers also have a parameter called the *stride*, which is used to control how much the dimensions of the output matrix are reduced. The stride defines how much we move the kernel at each step in the algorithm. In the above example, we used a stride of one, since we only moved the pooling kernel by one value at each step. The resulting matrix had two fewer values in each dimension (6x6 => 4x4). Here is the same example, but with a stride value of three:




Next, we move the kernel to the right by three, the value specified by the stride.




Similarly, we move the kernel down by three and compute the last row:




With a stride of three, the dimensions of our matrix decreased significantly (6x6 => 2x2), while retaining all the information about where the edges exist in the image. This pooled layer could then be fed into a standard fully-connected neural network to determine if an image were a four (has vertical edges) or a two (no vertical edges).

## Warm-up

1. Given a size M x M picture as an input to a convolutional layer, and an N x N filter applied to that input, what will be the dimensions of the output? You may assume stride is 1, and use valid padding.

2. Now suppose you use a stride of S with the same layer (so same input and filter dimensions). What are the dimensions of the output now? Hint: you may need to use floor division and use valid padding).

3. Open the file "WarmUp_Question3_Pooling.ipynb" and complete the following code:
    a. Fill in the two blank lines of code inside the `maxPool` function. You may want to use the "`x // y`" operator for floor division.
    b. Write an average pooling function and output the resulting image. You should be able to use almost the exact same code as the `maxPool` function, but change one line.

## Main activity

$$M = \begin{bmatrix} -1 & 0 & -1 & -2 \\ 2 & 1 & -2 & -4 \\ 2 & 0 & 2 & 1 \\ -2 & 3 & 1 & -3 \end{bmatrix} \qquad F = \begin{bmatrix} 1 & 0 \\ -1 & 2 \end{bmatrix}$$

1. Convolutional Layer
    a. Given the input matrix M and the filter F, what is the output of the convolutional layer when using a stride of 1 and valid-padding?
    b. Using the same matrices M and F and valid-padding, what is the output when using a stride of 2?

2. Apply ReLu activation function to your answer from part 1.a). What is the resulting matrix?

3. Pooling Layer
    a. Apply a max-pooling layer with a 2x2 pooling kernel and a stride of 1 to your answer from part 1.a). What is the resulting matrix?
    b. Do the same thing as in part 3.a), except use an average-pooling layer instead of a max-pooling layer to your answer from 1.a) (still 2x2 kernel with stride of 1). What is the resulting matrix? (Do not skip the factor of ¼)

4. Open and run the code in the file "Main_Activity_Question4.ipynb" to answer the following questions:
    a. What kind of filter is `filter1`? In other words, what is it doing to the original Bucky photo?
    b. How does changing the value of the stride parameter affect the output of the convolutional layer?
    c. Add another convolutional layer with a diagonal line filter (hint: use an identity matrix for a filter). What does the Bucky image look like now?
    d. Each convolutional layer can have many filters applied to them, and each filter has their own corresponding bias. Edit the `conv()` function to accept multiple filters and biases for each filter. The bias is added to the resulting convolution of the input and filter. Apply at least three filters (and biases) of your choice to the first convolutional layer and output your resulting Bucky's (Note: you will need to output a separate image for each filter)

5. This last file puts together all of the layers we learned into a small but complete CNN structure. Our CNN consists of 1 convolutional layer, an ReLu activation layer, a max pooling layer, and a fully connected neural network with a single hidden layer. Note that our implementation of CNN is incomplete, as true CNNs have multiple instances of the layers mentioned, backpropagate all the way back to the first layer instead of just through the neural network, and typically have lower

error rates. This CNN does hand-written digit recognition using the scikit-learn database. Print `show_digit(digits.images[x])` where 'x' is an index between 0 and 999 to see one of the 1000 8x8 handwritten digits being used. You will see that some of them are even hard for us to classify just by looking at them!

Open and run the code in "Main_Activity_Question5.ipynb". How does the error rate change when we specify different filters and different stride amounts? Note that the single layer neural network uses stochastic gradient descent, so error values may differ slightly between runs due to the stochastic nature.

## References


Bernet, C. (n.d.). Handwritten Digit Recognition with scikit-learn - The Data Frog. Retrieved April 29, 2020, from https://thedatafrog.com/handwritten-digit-recognition-scikit-learn

Dyer, C. (2019). Neural Networks [CS 540 PowerPoint slides]. University of Wisconsin-Madison. http://pages.cs.wisc.edu/~dyer/cs540/notes/09_neuralNets.pdf

Elia, E. (2019, July 28). A Guide to Building Convolutional Neural Networks from Scratch. Towards Data Science. Retrieved April 29, 2020, from https://towardsdatascience.com/a-guide-to-convolutional-neural-networks-from-scratch-f1e3bfc3e2de

He, K., Zhang, X., Ren, S. (2016). Deep Residual Learning for Image Recognition. IEEE Conference on Computer Vision and Pattern Recognition (CVPR), pp. 770-778. 10.1109/CVPR.2016.90

Howard, A., Zhang, C., and Horvitz, E. (2017), "Addressing bias in machine learning algorithms: A pilot study on emotion recognition for intelligent systems," IEEE Workshop on Advanced Robotics and its Social Impacts (ARSO), pp. 1-7.

Krizhevsky, A., Sutskever, I., and Hinton, G. (2017). ImageNet classification with deep convolutional neural networks. Commun. ACM 60, 6, 84–90. DOI:https://doi.org/10.1145/3065386

LeCun, Y., Bottou, L., Bengio, Y., Haffner, P. (1998). Gradient-based learning applied to document recognition. Proceedings of the IEEE, vol. 86, no. 11, pp. 2278-2324. 10.1109/5.726791

Saha, S. (2018, December 15). A Comprehensive Guide to Convolutional Neural Networks. Towards Data Science. Retrieved April 29, 2020, from https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53

# Appendix

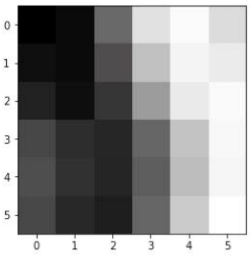
### Solutions to Warm-Up Activities

1. Output is a square matrix of size (M - N) + 1
2. Output is a square matrix of size ⌊(M - N) / S⌋ + 1 (⌊x⌋ means "floor x")
3. a)
```
H_out = (H - Hp) // stride + 1
W_out = (W - Wp) // stride + 1
```
   b)
```
def avgPool(x, Hp, Wp, stride):
    # get output dimensions
    H, W = x.shape
    H_out = (H - Hp) // stride + 1
    W_out = (W - Wp) // stride + 1
    out = np.zeros((H_out, W_out))

    # pool
    for i in range(H_out):
        for j in range(W_out):
            out[i, j] = np.mean(x[i*stride:i*stride+Hp,
j*stride:j*stride+Wp]) # only line different from maxPool

    return out
```

Output of average pooling should look something like this (pending parameters used):




### Solution to Main Activity

1. a) $\begin{bmatrix} -1 & -5 & -7 \\ 0 & 5 & -2 \\ 10 & -1 & -5 \end{bmatrix}$

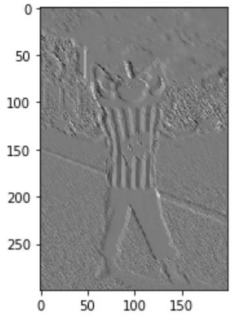
   b) $\begin{bmatrix} -1 & -7 \\ 10 & -5 \end{bmatrix}$

2. $\begin{bmatrix} 0 & 0 & 0 \\ 0 & 5 & 0 \\ 10 & 0 & 0 \end{bmatrix}$

3. a) $\begin{bmatrix} 5 & 5 \\ 10 & 5 \end{bmatrix}$

   b) $\begin{bmatrix} -1/4 & -9/4 \\ 7/2 & -3/4 \end{bmatrix}$

4. a) "`Filter1`" is a vertical edge detection filter, so the vertical edges on Bucky are more emphasized than non-vertical ones..
   b) As you increase the stride, time to convolve decreases, the output image shrinks in size, and the output image becomes more pixelated.

   c) 

   Note: image may vary slightly depending on filter size and stride

   d) Answers can vary, but this is one implementation:

```
def conv(x, filt, bias, stride, pad):
  """
  - H: height (number of rows)
  - W: width (number of columns)
  - F: number of filters
  - bias: array of biases to apply to each filter
  - stride: stride value to apply to each image
  - pad: number of rows and columns to zero-pad around image
  """
  H, W = x.shape
  F, HH, WW = filt.shape # need to pad matrices to be same dimension with this implementation
  # get output dimensions
  H_out = 1 + (H + 2 * pad - HH) // stride
  W_out = 1 + (W + 2 * pad - WW) // stride
  out = np.zeros((F, H_out, W_out))
  pad_widths = ((pad,), (pad,))
```

```
      xpad = np.pad(x, pad_widths, 'constant')
      Hpad, Wpad = xpad.shape
      # perform convolution
      for f in range(F):
         for i in range(0, Hpad-(HH-1), stride):
            for j in range(0, Wpad-(WW-1), stride):
                  prod = np.sum(np.multiply(filt[f], xpad[i:i+HH,
j:j+WW]))
                 out[f, int(i/stride), int(j/stride)] = prod +
bias[f]

      return out
```

5. Higher stride values increase the error, and generally larger and less filters increase the error (and vice versa).