Introduction to Deep Learning

Lecture 6
Convolutional Filters

Ian J. Watson

University of Seoul

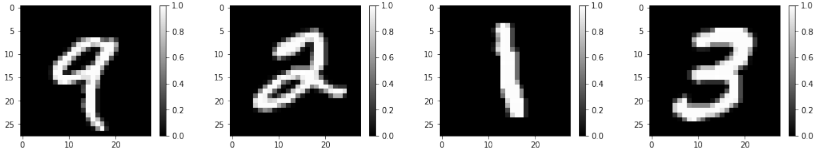
University of Seoul Graduate Course 2020

KOREA RESEARCH FELLOWSHIP
해외 우수신진연구자 유치사업

Housekeeping

- Some references for today
  - Official pytorch tutorials
  - Pytorch tutorials by yunjey, from beginning to advanced
  - Deep Learning Book on ConvNets
  - MIT Intro to Deep Learning Lecture on ConvNets
- Today's url
  - `https://git.io/2020deep06`
- Email if you need help with any of this!

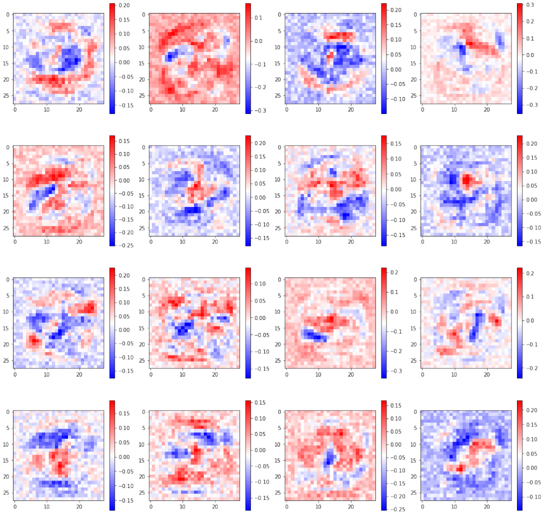
What do we want a neural network to do?



- Thinking of our mnist dataset from last week and a 2 layer hidden network, we might expect/want the network to piece together a structure like this
  - In the first layer, find loops and lines in the various parts of the image
  - In the second layer, see if they combine together (fire at the same time), in certain combinations
- In the example, theres a "loop" a the top connected to a "line" for 9, a similar "loop" at the bottom but connect to a "hook", a "line" similar to that in 9 for the 1, and some "hooks" for the 3
- Is the network structured this way? I.e. are there "loop", "hook", "line" finders

idea from 3blue1brown on neural nets

What does it really do?

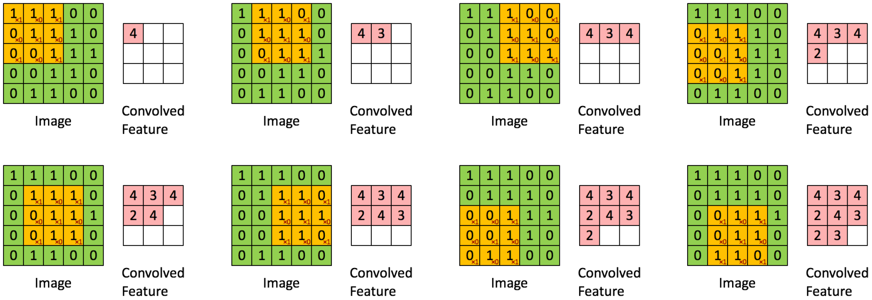


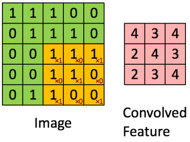
- Plot of the weights from each pixel connection in the first layer of a 2 hidden layer model from MNIST (red means this pixel should be fired to fire this node, blue means that if the pixel is fired the node is suppressed)
- Maybe recognizable structures, but lots of fairly random swirls
- One issue with the network is that structures like loops and lines need to be fit, but these can appear in different places on the image
- So, we don't have a "loop finder" node, we need to find loops in any of many different points on the image, leading to a jumble of weights
- Can we change our network so "loop finder" structures are possible?

Convolutional Layers

- A convolution layer is a connection between one layer and the next in a NN with a very specific structure:
  - Typically, it works with a 3d input like an image: channels (red, green, blue), width, height
  - It contains a **kernel** or **filter**, which is a 3d block sized $channel \times n \times m$, n and m are user-specifed, with each element of the block a weight to be set in training
  - The outputs consists of all $n \times m$ *convolutions* of the filter with the image, creating a new one-channel image
    - Discrete convolution, meaning each element of the kernel is multiplied with a pixel in (one channel of) the image, and all are summed together
  - The output of the filter is passed through an activation function, the same as the usual fully-connected layer
- A single convolutional layer generally consists of many convolutional filters, each filter giving one layer in the output
- Networks with convolutional layers are Convolutional Neural Networks: CNN

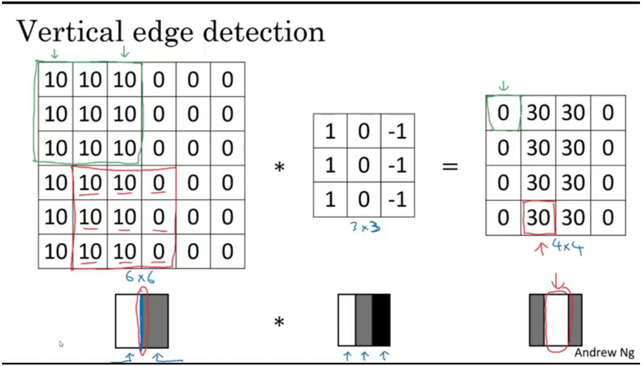
Convolutional Filters In Pictures





- A filter sliding over the image builds up the output layer, each output is sum of filter elements multiplied by image pixels
- The same filter is used for each pixel, the weights are learnt during training (as well as an output bias)

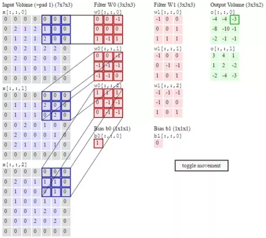
Example Filter



- As an example, here is a 3x3 filter for detecting vertical edges
- The opposing plus and minus sides cancel in a block of color
- At an edge, the filter is either highly positive (white to left of edge), or negative (white to right of edge)
- What would a horizontal edge detector look like?

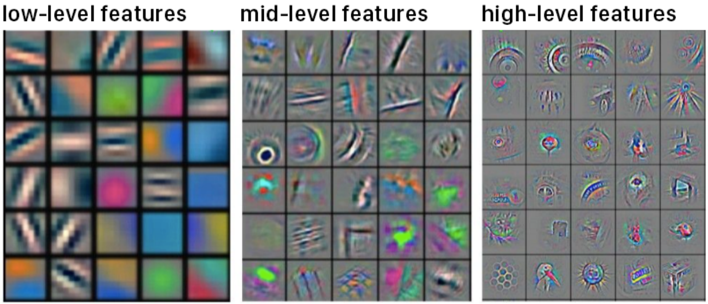
Andrew Ng lecture by way of `https://kharshit.github.io/blog/2018/12/14/filters-in-convolutional-neural-networks`

Multiple Filter Outputs



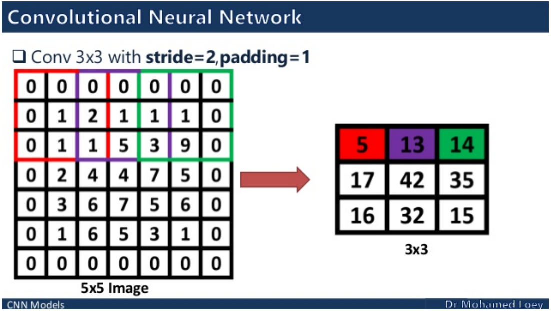
- When multiple filters are used in a single layer, they have the same width and height, so they can be put together in a single output as *channels* $\times$ *width* $\times$ *height*
- This is exactly the image structure which was the input to the network
- This means this convolutional structure can be used several times in series
    - Each successive layer effectively sees a larger part of the image, since each pixel in the output of one layer is from several pixels
- The image shows that a 3-channel input needs filters with a 3x3x3 block, and 2 filters produce a 2 channel output

Filters Over Several Input Layers



- Convolutional layers are typically built up one after the other
- The idea is that features get *built up*, at low levels, you might have edge detectors, later layers use these edges to build up structure, and by high levels recognizable objects are being searched for
  - These images are made by doing reverse gradient descent on the network, i.e. updating the image pixels themselves, trying to make the image "light up" (set node output high) a particular node
- Networks these days can contain *hundreds* of these layers
  - This is the meaning of *deep* in deep learning

Strides and Padding



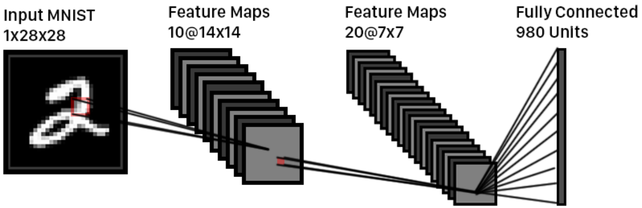
- When sliding across the image, you can move the filter more than 1 pixel at a time, this is the *stride*
    - By default its just 1, ie sliding the image
- The filter will reduce the size of the image (can only fit in so many 3x3 blocks), you can *pad* the image (with zeros, or copying the outer variables) to keep the outputs the same size
- Can also use different strides or pads in the vertical and horizontal directions

Pooling Layers

$$\begin{pmatrix} 1 & 2 & 3 & 4 \\ 2 & 3 & 4 & 5 \\ 4 & 3 & 2 & 1 \\ 5 & 4 & 3 & 2 \end{pmatrix} \rightarrow^{2x2}_{max} \begin{pmatrix} 3 & 5 \\ 5 & 3 \end{pmatrix}$$

- We may want to reduce the size of the images flowing through the network for computational and conceptual efficiency reasons
  - As we add filters, we should be building up higher level features, which are less localized on the image
  - Another way to say this is we want to *downsample* the image
- We can reduce the image through *pooling*, applying an operation on each $n \times n$ patch of the image (leaping **not** sliding)
- A typical use is max pooling, we could find the maximum of each patch of the image
- Here, we apply a 2×2 max pooling to reduce a 4×4 matrix to 2×2
- Another typical operation is to take the *average* of each patch

Structuring a Network with Convolutional Layers



- The basic CNN consists of several convolutional layers, followed by "squashing" the output of the last convolution into a regular 1d node structure, after which the fully connected layers of a normal NN can be used
- So the idea is, the convolutional layers search for particular high level "features", then the output is decided by which features do or do not exist in the network

Image from `https://twopointseven.github.io/2017-10-29/cnn/`

Some Benefits of The Convolutional Neural Network

- Fewer parameters than a fully connected network
  - Parameters for a cxhxw image fully connected to n nodes: cxhxwxn + n
  - Parameters for a cxhxw image convolutional to n mxm filters (no padding/stride 1): cxmxmxn + n
  - If our filter size is smaller than the image, much, much fewer parameters, and independent of input height, width
  - Fewer parameters is better for overtraining
- The sliding connections mean the network can learn features independent of position
  - A fully connected layer would need to learn what a 'hand' or an 'eye' looks like independently everywhere it could be in the image
  - This *parameter sharing* between parts of the image means that the network can learn more robust features

Convolutional Filters in pytorch

- torch.nn.Conv2d provides a convolutional filter, you tell it:
  - The number of input channels
  - The number of output channels
  - The size of the filter (can be a number for nxn or a 2-tuple for nxm)
  - Optionally, you can change the stride and the padding
- The filters take in tensors of rank 4, with shape: `(number of images, number of channels, height of image, width of image)` (pytorch always assumes you're processing multiple images)
- The output is also a rank 4 tensor, with the number of output channels changed, and the height and width can be expanded or contracted by changing the stride and padding

```
# convolutional filter from 1 -> 2 channels, with 3x3 filter
conv_filter = torch.nn.Conv2d(1,2,3)
conv_filter(torch.tensor([ [ [[1,1,1],[1,1,1],[1,1,1.]] ] ]))
```

```
tensor([[[[-0.2259]],
         [[-0.1640]]]], grad_fn=<MkldnnConvolutionBackward>)
```

Pooling Layers in pytorch

- Similar to Conv2d, there is `torch.nn.MaxPool2d` and `torch.nn.AvgPool2d` to max and average pooling respectively,
- They only need to be given the filter size, and have similar input/output shapes (rank-4 tensors everywhere)

```
pool = torch.nn.MaxPool2d(2)
pool(torch.tensor([ [ [[1,2,3,4],[1,1,1,1],
                       [1,1,1.,1],[4,5,6,7]] ] ]))
```

```
tensor([[[[2., 4.],
          [5., 7.]]]])
```

```
pool = torch.nn.AvgPool2d(2)
pool(torch.tensor([ [ [[1,2,3,4],[1,1,1,1],
                       [1,1,1.,1],[4,5,6,7]] ] ]))
```

```
tensor([[[[1.2500, 2.2500],
          [2.7500, 3.7500]]]])
```

Building a Network

- Networks will at some point need to go from processing 2d images with multiple channels, to a discrete probability distribution (if we are making a classifier)
- You can insert a `view` into the `forward` function to adjust the output nodes into a 1d line (-1 at the front so it automatically sizes to any number of images in the input)
- Here is a simple CNN for MNIST with 1 convolutional layer, which is reshaped and then connected to the 10 category output layer

```
class SimpleCNN(torch.nn.Module):
  def __init__(self):
    super(SimpleCNN, self).__init__()
    self.conv = torch.nn.Conv2d(1,6,5) # 5x5 filter, no padding
    self.fc = torch.nn.Linear(6*24*24,10)
  def forward(self, x):
    x = torch.tanh(self.conv(x))
    x = self.fc(x.view(-1, 6*24*24))
    return x
```

Exercises

We will train 2 convolutional networks in pytorch.