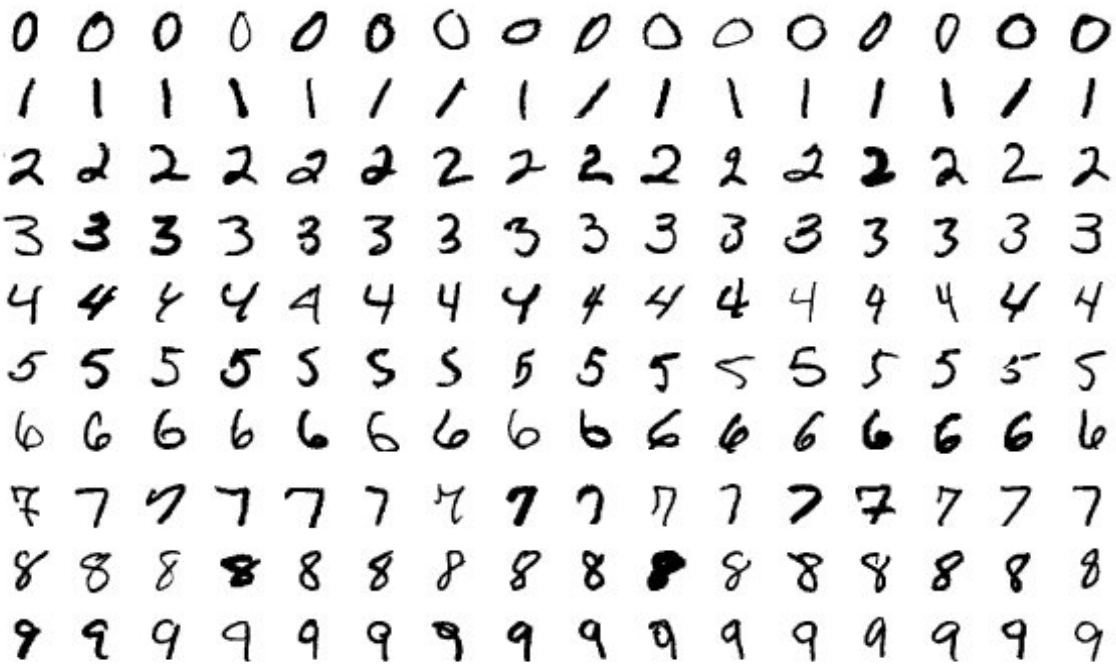
Convolutional Neural Network

Dataset



Dataset is MNIST handwritten digits(with “jpg” extension). Each image in the MNIST dataset is 28x28 and contains a centered.

Dataset images are 3 channel RGB images.Transforming images from 3 channels to 1 channel makes it easier to work with them.OpenCV(Open source computer vision) library is for that operations.

You could treat each image as a 28 x 28 = 784-dimensional vector, feed that to a 784-dim input layer, stack a few hidden layers, and finish with an output layer of 10 nodes, 1 for each digit.

This would only work because the MNIST dataset contains small images that are centered, so we wouldn’t run into the aforementioned issues of size or shifting. Most real-world image classification problems aren’t this easy.

First digit of name is label for image.

For example:

“0img_5.jpg” represents number 0

“5img_62.jpg” represents number 5

LAYERS

1) Convolutional layers

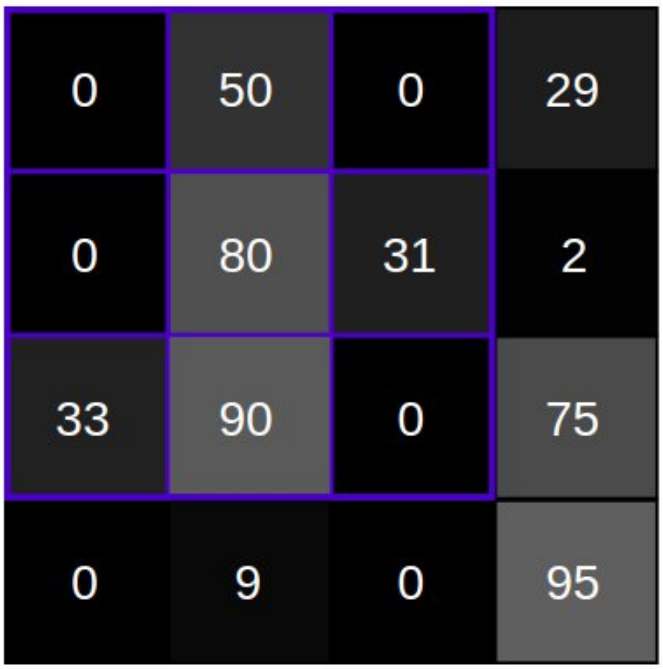
Convolutional layers, a.k.a. Conv layers, which are based on the mathematical operation of convolution. Conv layers consist of a set of filters, which you can think of as just 2d matrices of numbers. Here’s an example 3x3 filter:

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

We can use an input image and a filter to produce an output image by convolving the filter with the input image. This consists of

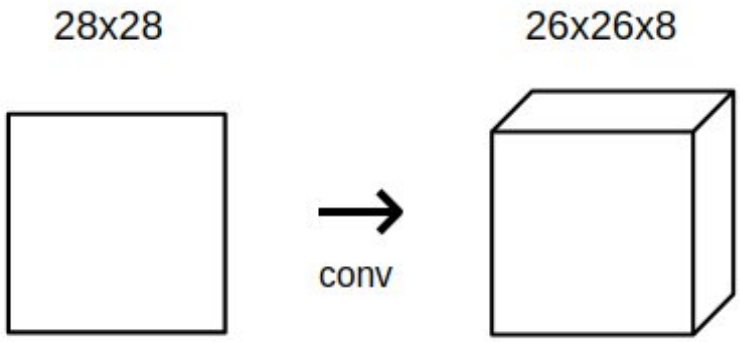
1. Overlaying the filter on top of the image at some location.
2. Performing element-wise multiplication between the values in the filter and their corresponding values in the image.
3. Summing up all the element-wise products. This sum is the output value for the destination pixel in the output image.

4. Repeating for all locations.



Finally, we place our result in the destination pixel of our output image. Since our filter is overlayed in the top left corner of the input image, our destination pixel is the top left pixel of the output image:

For our MNIST CNN, we'll use a small conv layer with 8 filters as the initial layer in our network. This means it'll turn the 28x28 input image into a 26x26x8 output volume:

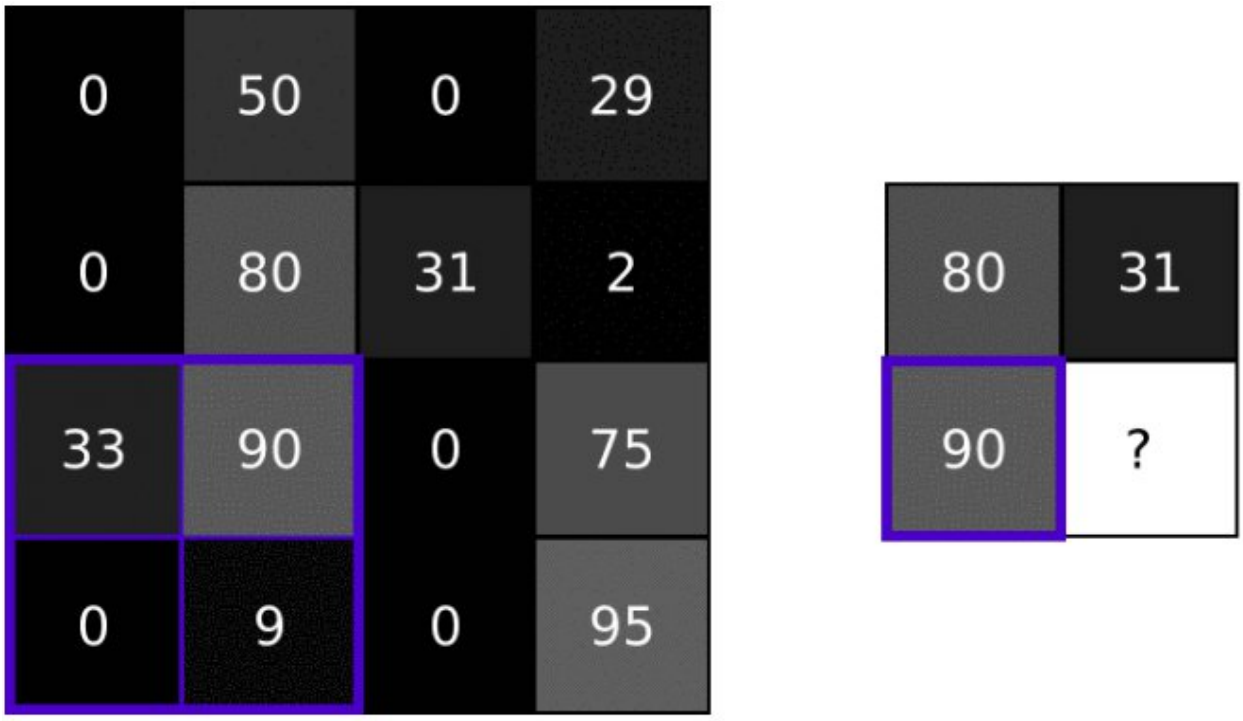


Each of the 8 filters in the conv layer produces a 26x26 output, so stacked together they make up a 26x26x8 volume. All of this happens because of 3 × 3 (filter size) × 8 (number of filters) = only 72 weights.

2) Pooling layers

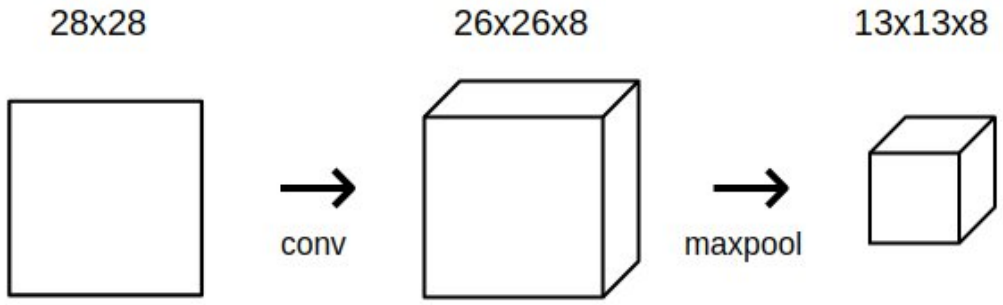
Neighboring pixels in images tend to have similar values, so conv layers will typically also produce similar values for neighboring pixels in outputs. As a result, much of the information contained in a conv layer's output is redundant. For example, if we use an edge-detecting filter and find a strong edge at a certain location, chances are that we'll also find relatively strong edges at locations 1 pixel shifted from the original one. However, these are all the same edge! We're not finding anything new.

Pooling layers solve this problem. All they do is reduce the size of the input it's given by (you guessed it) pooling values together in the input. The pooling is usually done by a simple operation like max, min, or average. Here's an example of a Max Pooling layer with a pooling size of 2:



To perform *max* pooling, we traverse the input image in 2x2 blocks (because pool size = 2) and put the *max* value into the output image at the corresponding pixel. That's it!

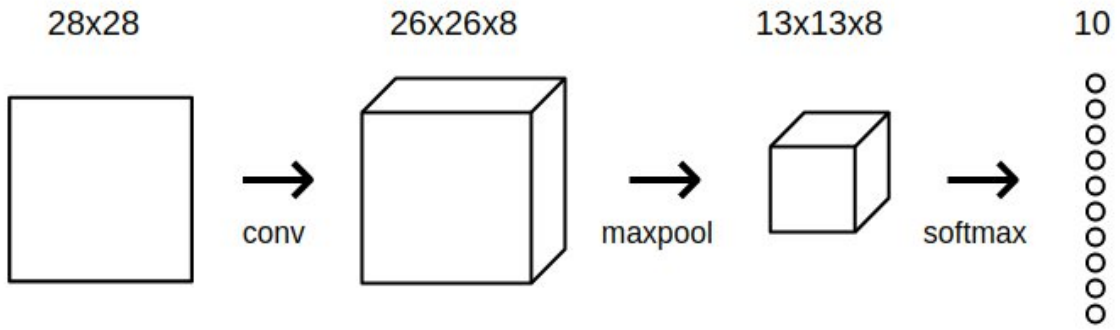
Pooling divides the input's width and height by the pool size. For our MNIST CNN, we'll place a Max Pooling layer with a pool size of 2 right after our initial conv layer. The pooling layer will transform a 26x26x8 input into a 13x13x8 output:



3) Softmax layer

To complete our CNN, we need to give it the ability to actually make predictions. We'll do that by using the standard final layer for a multiclass classification problem: the Softmax layer, a fully-connected (dense) layer that uses the Softmax function as its activation.

We'll use a softmax layer with 10 nodes, one representing each digit, as the final layer in our CNN. Each node in the layer will be connected to every input. After the softmax transformation is applied, the digit represented by the node with the highest probability will be the output of the CNN!



What softmax really does is help us quantify how sure we are of our prediction, which is useful when training and evaluating our CNN. More specifically, using softmax lets us

use cross-entropy loss, which takes into account how sure we are of each prediction. Here's how we calculate cross-entropy loss:

$$L = -\ln(p_c)$$

where c is the correct class (in our case, the correct digit), p_c is the predicted probability for class c, and ln is the natural log. As always, a lower loss is better. For example, in the best case, we'd have

$$p_c=1,\ L=-\ln(1)=0$$

Training Overview

Training a neural network typically consists of two phases:

1. A forward phase, where the input is passed completely through the network.
2. A backward phase, where gradients are backpropagated (backprop) and weights are updated.

During the forward phase, each layer will cache any data (like inputs, intermediate values, etc) it'll need for the backward phase. This means that any backward phase must be preceded by a corresponding forward phase.

During the backward phase, each layer will receive a gradient and also return a gradient. It will receive the gradient of loss with respect to its *outputs* ($\partial L/\partial \text{out}$) and return the gradient of loss with respect to its *inputs* ($\partial L/\partial \text{in}$).

1) Backprop Softmax

We'll start our way from the end and work our way towards the beginning, since that's how backprop works. First, recall the cross-entropy loss:

$$L = -\ln(p_c)$$

where p_c is the predicted probability for the correct class c (in other words, what digit our current image *actually* is).

The first thing we need to calculate is the input to the Softmax layer's backward phase, $\partial L/\partial outs$, where *outs* is the output from the Softmax layer: a vector of 10 probabilities. This is pretty easy, since only p_i shows up in the loss equation:

$$\frac{\partial L}{\partial out_s(i)} = \begin{cases} 0 & \text{if } i \neq c \\ -\frac{1}{p_i} & \text{if } i = c \end{cases}$$

c is the correct class

We cache 3 things here that will be useful for implementing the backward phase:

- The input's shape *before* we flatten it (13 x 13 x 8).
- The input *after* we flatten it.
- The totals, which are the values passed in to the softmax activation.

With that out of the way, we can start deriving the gradients for the backprop phase. We've already derived the input to the Softmax backward phase: $\partial L/\partial out_s$. One fact we can use about $\partial L/\partial out_s$ is that *it's only nonzero for c, the correct class*. That means that we can ignore everything but $out_s(c)$!

First, let's calculate the gradient of $out_s(c)$ with respect to the totals (the values passed in to the softmax activation). Let t_i be the total for class i. Then we can write $out_s(c)$ as:

$$out_s(c) = \frac{e^{t_c}}{\sum_i e^{t_i}} = \frac{e^{t_c}}{S}$$

where $S = \sum_i e^{t_i}$.

Now, consider some class *k* such that k != *c*. We can rewrite $out_s(c)$ as:

$$out_s(c) = e^{t_c} S^{-1}$$

Chain Rule to derive:

$$\begin{aligned}
\frac{\partial out_s(c)}{\partial t_k} &= \frac{\partial out_s(c)}{\partial S}\left(\frac{\partial S}{\partial t_k}\right) \\
&= -e^{t_c} S^{-2}\left(\frac{\partial S}{\partial t_k}\right) \\
&= -e^{t_c} S^{-2}\left(e^{t_k}\right) \\
&= \boxed{\frac{-e^{t_c} e^{t_k}}{S^2}}
\end{aligned}$$

Remember, that was assuming k != *c*. Now let's do the derivation for *c*, this time using Quotient Rule:

$$\begin{aligned}
\frac{\partial out_s(c)}{\partial t_c} &= \frac{Se^{t_c} - e^{t_c}\frac{\partial S}{\partial t_c}}{S^2} \\
&= \frac{Se^{t_c} - e^{t_c} e^{t_c}}{S^2} \\
&= \boxed{\frac{e^{t_c}\left(S - e^{t_c}\right)}{S^2}}
\end{aligned}$$

Remember how $\partial L/\partial out_s$ is only nonzero for the correct class, *c*? We start by looking for *c* by looking for a nonzero gradient in d_L_d_out. Once we find that, we calculate the gradient $\partial out_s(i)/\partial t$ (d_out_d_totals) using the results we derived above:

$$\frac{\partial out_s(k)}{\partial t} = \begin{cases} \frac{-e^{t_c} e^{t_k}}{S^2} & \text{if } k \neq c \\ \frac{e^{t_c}(S - e^{t_c})}{S^2} & \text{if } k = c \end{cases}$$

We ultimately want the gradients of loss against weights, biases, and input:

- We'll use the weights gradient, $\partial L/\partial w$, to update our layer's weights.
- We'll use the biases gradient, $\partial L/\partial b$, to update our layer's biases.
- We'll return the input gradient, $\partial L/\partial input$, from our backprop() method so the next layer can use it. This is the return gradient we talked about in the Training Overview section!

To calculate those 3 loss gradients, we first need to derive 3 more results: the gradients of *totals* against weights, biases, and input. The relevant equation here is:

$$t = w * input + b$$

Gradients:

$$\frac{\partial t}{\partial w} = input$$

$$\frac{\partial t}{\partial b} = 1$$

$$\frac{\partial t}{\partial input} = w$$

Putting everything together:

$$\frac{\partial L}{\partial w} = \frac{\partial L}{\partial out} * \frac{\partial out}{\partial t} * \frac{\partial t}{\partial w}$$

$$\frac{\partial L}{\partial b} = \frac{\partial L}{\partial out} * \frac{\partial out}{\partial t} * \frac{\partial t}{\partial b}$$

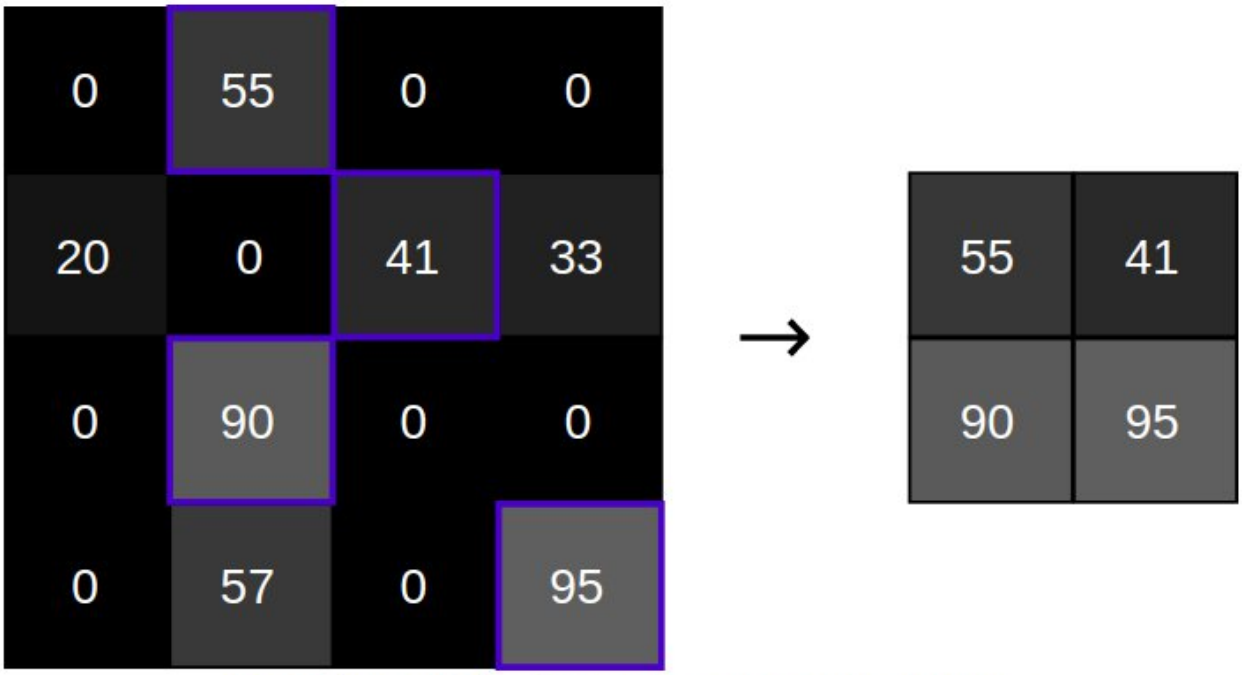
$$\frac{\partial L}{\partial input} = \frac{\partial L}{\partial out} * \frac{\partial out}{\partial t} * \frac{\partial t}{\partial input}$$

2) Backprop Pooling

A Max Pooling layer can't be trained because it doesn't actually have any weights, but we still need to implement a backprop() method for it to calculate gradients. We'll start by adding forward phase caching again. All we need to cache this time is the input:

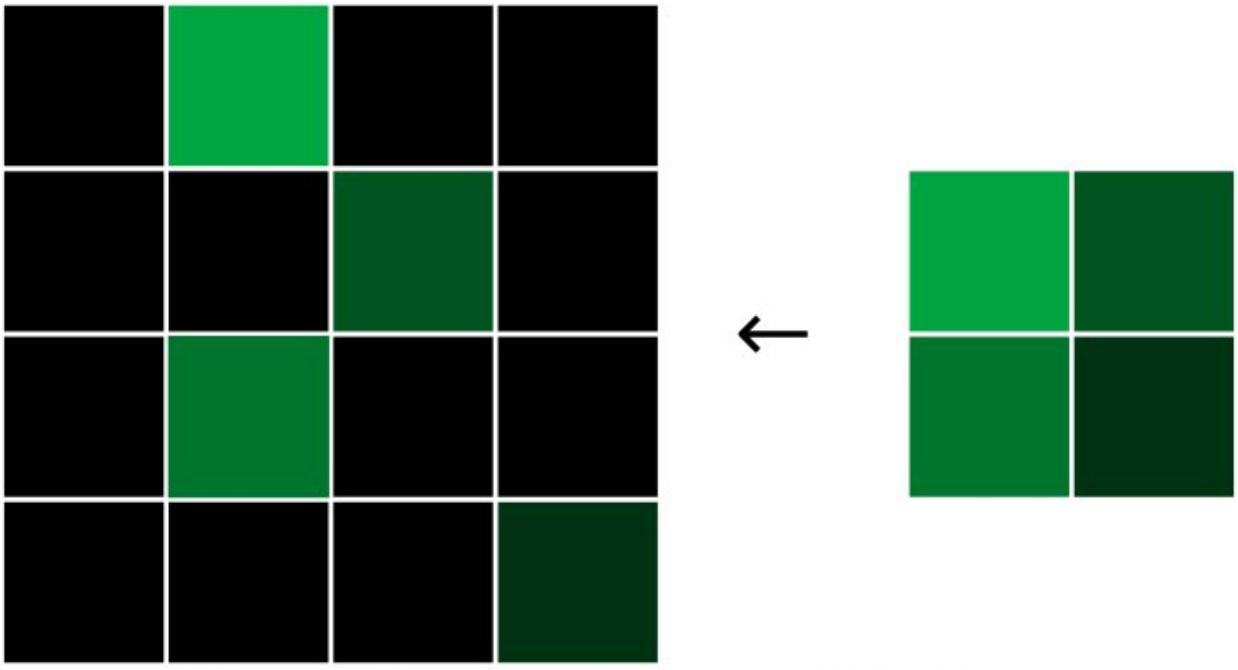
During the forward pass, the Max Pooling layer takes an input volume and halves its width and height dimensions by picking the max values over 2x2 blocks. The backward pass does the opposite: we'll double the width and height of the loss gradient by assigning each gradient value to where the original max value was in its corresponding 2x2 block.

Here's an example. Consider this forward phase for a Max Pooling layer:



An example forward phase that transforms a 4x4 input to a 2x2 output

The backward phase of that same layer would look like this:



An example backward phase that transforms a 2x2 gradient to a 4x4 gradient

Each gradient value is assigned to where the original max value was, and every other value is zero.

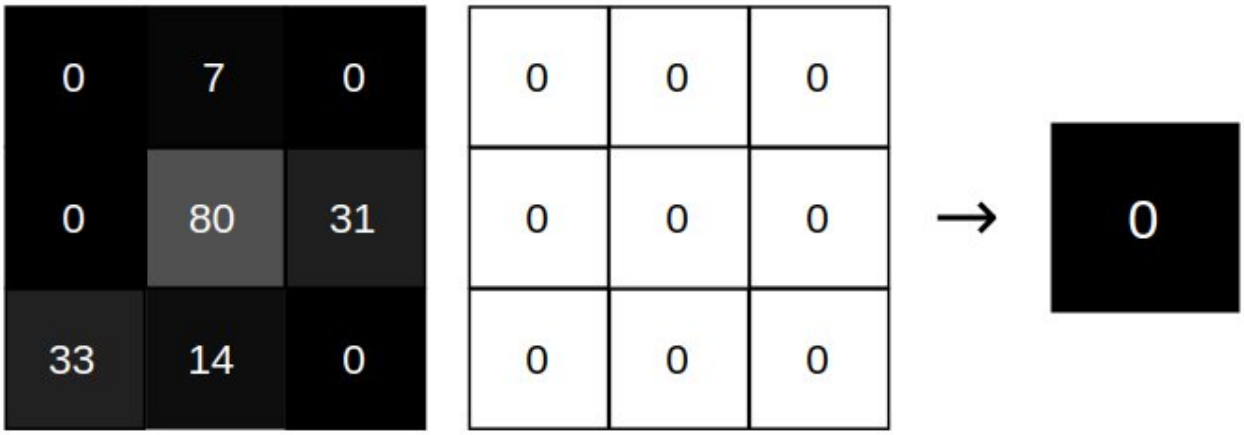
Why does the backward phase for a Max Pooling layer work like this? Think about what $\partial L/\partial inputs$ intuitively should be. An input pixel that isn't the max value in its 2x2 block would have *zero* marginal effect on the loss, because changing that value slightly wouldn't change the output at all! In other words, $\partial L/\partial input=0$ for non-max pixels. On the other hand, an input pixel that *is* the max value would have its value passed through to the output, so $\partial output/\partial input=1$, meaning $\partial L/\partial input=\partial L/\partial output$.

For each pixel in each 2x2 image region in each filter, we copy the gradient from d_L_d_out to d_L_d_input if it was the max value during the forward pass.

3) Backprop Convolution

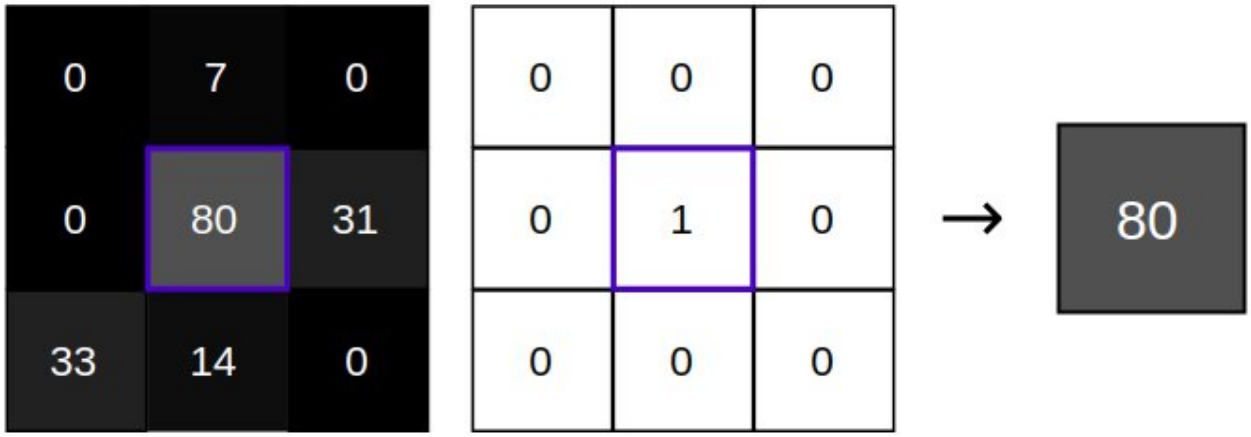
We're primarily interested in the loss gradient for the filters in our conv layer, since we need that to update our filter weights. We already have $\partial L/\partial out$ for the conv layer, so we just need $\partial out/\partial filters$. To calculate that, we ask ourselves this: how would changing a filter's weight affect the conv layer's output?

Here's a super simple example to help think about this question:



A 3x3 image (left) convolved with a 3x3 filter (middle) to produce a 1x1 output (right)

We have a 3x3 image convolved with a 3x3 filter of all zeros to produce a 1x1 output. What if we increased the center filter weight by 1? The output would increase by the center image value, 80:



Similarly, increasing any of the other filter weights by 1 would increase the output by the value of the corresponding image pixel! This suggests that the derivative of a specific output pixel with respect to a specific filter weight is just the corresponding image pixel value. Doing the math confirms this:

$$\begin{aligned}\text{out(i, j)} &= \text{convolve(image, filter)} \\ &= \sum_{x=0}^{3}\sum_{y=0}^{3} \text{image}(i+x, j+y) * \text{filter}(x, y)\end{aligned}$$

$$\frac{\partial \text{out}(i,j)}{\partial \text{filter}(x,y)} = \text{image}(i+x, j+y)$$

We can put it all together to find the loss gradient for specific filter weights:

$$\frac{\partial L}{\partial \text{filter}(x,y)} = \sum_i \sum_j \frac{\partial L}{\partial \text{out}(i,j)} * \frac{\partial \text{out}(i,j)}{\partial \text{filter}(x,y)}$$

Run example

Epochs: 200.

Learn rate: 0.003.

```
----------Training Netork----------
Path----------/home/varuzhan/Desktop/CNN/trainset_10000/
Epoch 0 : Average Loss 1.16041 , Accuracy 76.24 %
Epoch 1 : Average Loss 0.626148 , Accuracy 86.57 %
Epoch 2 : Average Loss 0.510506 , Accuracy 88.2 %
Epoch 3 : Average Loss 0.453745 , Accuracy 89.15 %
Epoch 4 : Average Loss 0.418221 , Accuracy 89.79 %
Epoch 5 : Average Loss 0.39313 , Accuracy 90.16 %
Epoch 6 : Average Loss 0.374068 , Accuracy 90.56 %
Epoch 7 : Average Loss 0.358865 , Accuracy 90.81 %
Epoch 8 : Average Loss 0.346311 , Accuracy 91.03 %
Epoch 9 : Average Loss 0.335673 , Accuracy 91.27 %
Epoch 10 : Average Loss 0.326476 , Accuracy 91.53 %
```

```
Epoch 193 : Average Loss 0.14668 , Accuracy 96.13 %
Epoch 194 : Average Loss 0.146396 , Accuracy 96.15 %
Epoch 195 : Average Loss 0.146113 , Accuracy 96.14 %
Epoch 196 : Average Loss 0.145832 , Accuracy 96.14 %
Epoch 197 : Average Loss 0.145552 , Accuracy 96.15 %
Epoch 198 : Average Loss 0.145274 , Accuracy 96.15 %
Epoch 199 : Average Loss 0.144997 , Accuracy 96.15 %
----------Testing Netork----------
Path----------/home/varuzhan/Desktop/CNN/testset_1000/
Test results : Average Loss 0.22414 , Accuracy 93.1 %.
Images count 1000 , Correct predicted 931 , Wrong predicted 69
```

In only 10000 training steps, we went to 0.22414 loss and 93.1% accuracy

Hints

testset_1000.zip contains 1000 images for testing the network.

trainset_10000.zip contains 10000 images for training.

Compile command (Linux terminal) : g++ main.cpp normal_random.cpp direction_scan.cpp image.cpp convolution.cpp pooling.cpp softmax.cpp cnn.cpp -lopencv_core -lopencv_imgcodecs -o a.out