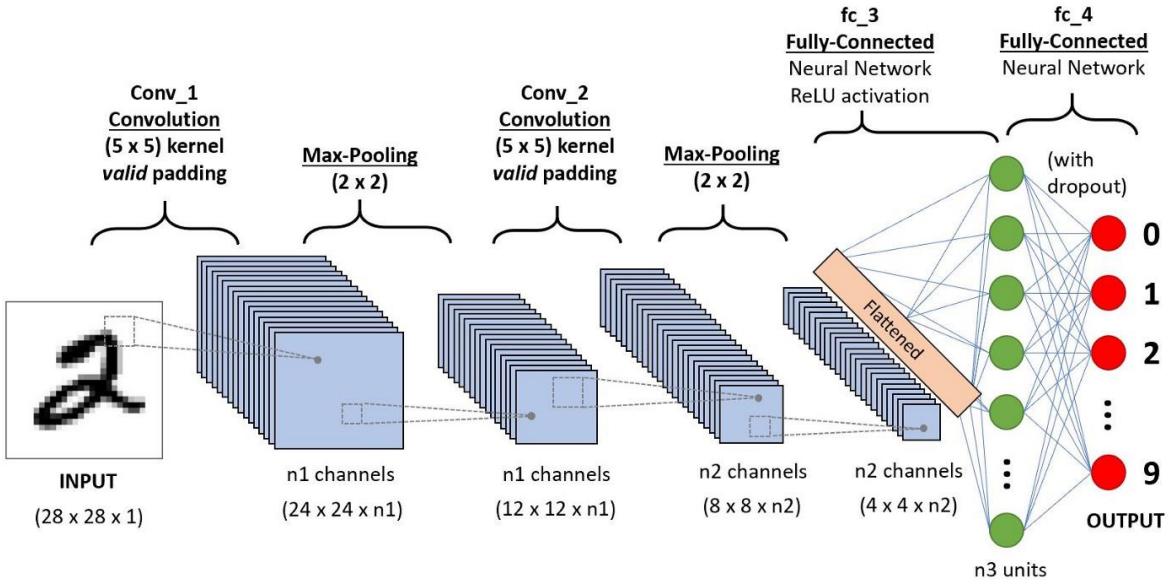
Assignment 2: Exercise 1 & 2

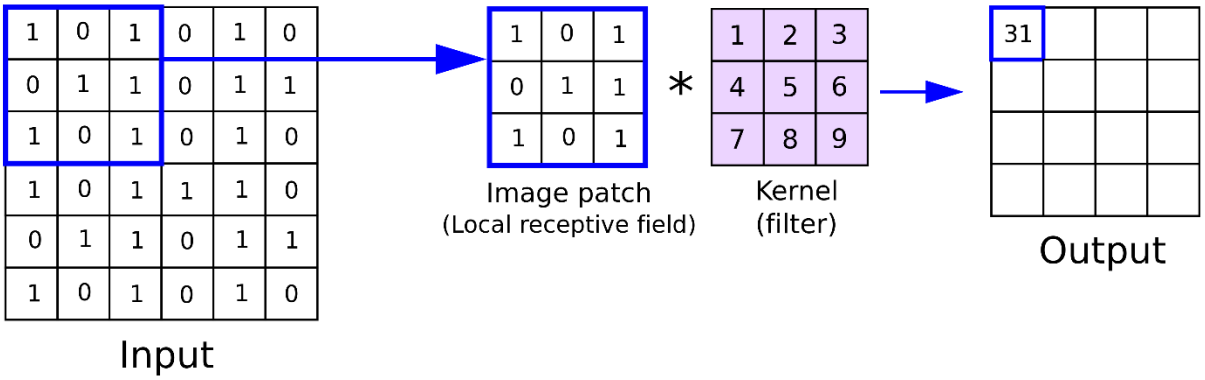
Nima Taheri, Mahdieh Sajedi Pour

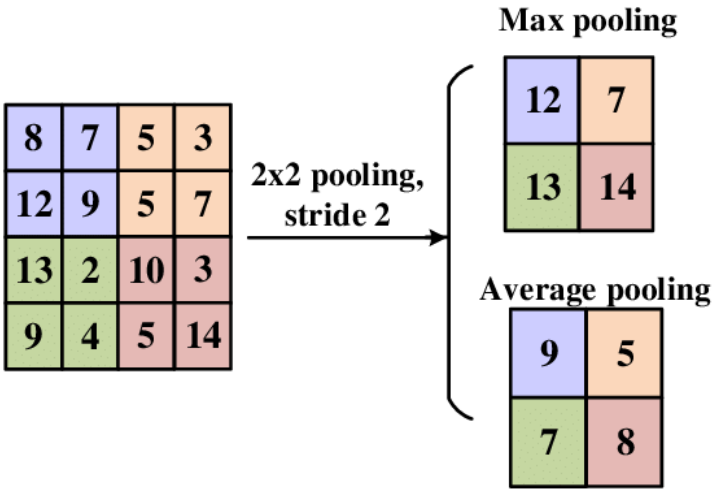
November 17, 2022

Exercise 1: Backpropagation in Convolutional Neural Network



In the picture above, we can see a convolutional neural network with pooling layers.
After performing convolution on the data, a pooling operation is performed, which gradually reduces the spatial size of the representation and thus reduces the amount of calculations and parameters in the network. In the end, we have a one-dimension vector and fed into a fully-connected MLP for processing.





In convolutional network above there are filters of size 3, with stride set to 1. Also, we can see two kinds of pooling layers; max pooling and average pooling.

Error:

$$E = \frac{1}{2} \sum_{p} (t_p - y_p)^2$$

The predicted outputs are mentioned as y_p and target values t_p. Learning will change the random weights such that y_p is as close as possible to t_p.

Backpropagation:

First let's discuss pooling layers. Pooling layers just reduce the amount of parameters and do nothing due to learn. In the forward propagation step, an N×N pooling block is reduced to a single value called the “winning unit”. So, in the backpropagation we know that the error is caused by the “winning unit”.

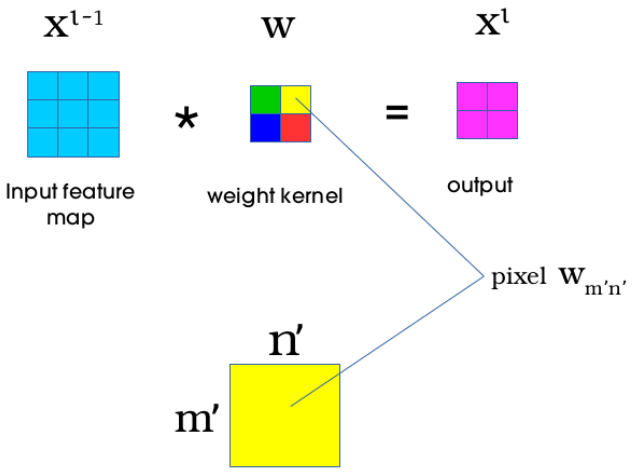
We must do the following:

For max-pooling: As the blocks expect "winning unit" don't have an effect on the error; the error is just assigned to where it comes from and other blocks are assigned zero.

For average pooling: The error is divided in N×N and assigned to the whole pooling layer.

In the convolutional layer there are two updates that must be done:

Weights: We need $\frac{\partial E}{\partial w^l_{m',n'}}$ to know how a single pixel w_m',n' affects the loss function E.



Because w_m', n' makes a contribution in all the products, it will affect all the elements in the output feature map.

Convolution between the input feature map of size H×W and the filter of size k_1×k_2 produces an output feature map of size (H−k1+1) × (W−k2+1) (if the stride is set to 1).

$$\frac{\partial E}{\partial w^l_{m',n'}} = \sum_{i=0}^{H-k_1}\sum_{j=0}^{W-k_2} \frac{\partial E}{\partial x^l_{i,j}} \frac{\partial x^l_{i,j}}{\partial w^l_{m',n'}}$$

$$= \sum_{i=0}^{H-k_1}\sum_{j=0}^{W-k_2} \delta^l_{i,j} \frac{\partial x^l_{i,j}}{\partial w^l_{m',n'}}$$

$x^l_{i,j}$ is equivalent to $\sum_m \sum_n w^l_{m,n} o^{l-1}_{i+m,j+n} + b^l$

Thus, we have:

$$\frac{\partial x^l_{i,j}}{\partial w^l_{m',n'}} = \frac{\partial}{\partial w^l_{m',n'}} \left(\sum_m \sum_n w^l_{m,n} o^{l-1}_{i+m,j+n} + b^l \right)$$

Then we have:

$$\frac{\partial x^l_{i,j}}{\partial w^l_{m',n'}} = \frac{\partial}{\partial w^l_{m',n'}} \left(w^l_{0,0} o^{l-1}_{i+0,j+0} + \cdots + w^l_{m',n'} o^{l-1}_{i+m',j+n'} + \cdots + b^l \right)$$

$$= \frac{\partial}{\partial w^l_{m',n'}} \left(w^l_{m',n'} o^{l-1}_{i+m',j+n'} \right)$$

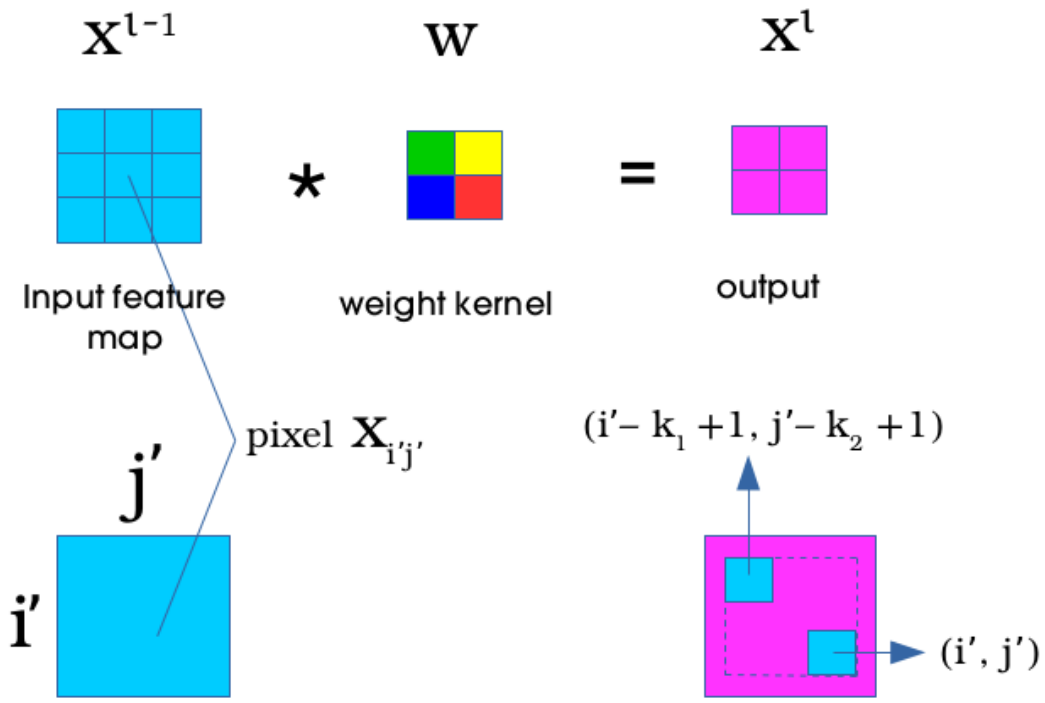
$$= o^{l-1}_{i+m',j+n'}$$

for all except the components where m=m′ and n=n′.

Substituting $\frac{\partial x^l_{i,j}}{\partial w^l_{m',n'}}$ in $\frac{\partial E}{\partial w^l_{m',n'}}$ will give us:

$$\frac{\partial E}{\partial w^l_{m',n'}} = \sum_{i=0}^{H-k_1} \sum_{j=0}^{W-k_2} \delta^l_{i,j} o^{l-1}_{i+m',j+n'}$$

Deltas: Here we need $\frac{\partial E}{\partial x^l_{i',j'}}$ to know how a single pixel x_i',j' affects the loss function E



The bounded region in the output is affected by x_i',j'.

Using chain rule, we have:

$$\frac{\partial E}{\partial x^l_{i',j'}} = \sum_{m=0}^{k_1-1} \sum_{n=0}^{k_2-1} \frac{\partial E}{\partial x^{l+1}_{i'-m,j'-n}} \frac{\partial x^{l+1}_{i'-m,j'-n}}{\partial x^l_{i',j'}}$$

$$= \sum_{m=0}^{k_1-1} \sum_{n=0}^{k_2-1} \delta^{l+1}_{i'-m,j'-n} \frac{\partial x^{l+1}_{i'-m,j'-n}}{\partial x^l_{i',j'}}$$

$x^{l+1}_{i'-m,j'-n}$ is equivalent to $\sum_{m'} \sum_{n'} w^{l+1}_{m',n'} o^l_{i'-m+m',j'-n+n'} + b^{l+1}$

$$\frac{\partial x^{l+1}_{i'-m,j'-n}}{\partial x^{l}_{i',j'}} = \frac{\partial}{\partial x^{l}_{i',j'}}\left(\sum_{m'}\sum_{n'} w^{l+1}_{m',n'} o^{l}_{i'-m+m',j'-n+n'} + b^{l+1}\right)$$
$$= \frac{\partial}{\partial x^{l}_{i',j'}}\left(\sum_{m'}\sum_{n'} w^{l+1}_{m',n'} f\left(x^{l}_{i'-m+m',j'-n+n'}\right) + b^{l+1}\right)$$

Then we have:

$$\frac{\partial x^{l+1}_{i'-m,j'-n}}{\partial x^{l}_{i',j'}} = \frac{\partial}{\partial x^{l}_{i',j'}}\left(w^{l+1}_{m',n'} f\left(x^{l}_{0-m+m',0-n+n'}\right) + \cdots + w^{l+1}_{m,n} f\left(x^{l}_{i',j'}\right) + \cdots + b^{l+1}\right)$$
$$= \frac{\partial}{\partial x^{l}_{i',j'}}\left(w^{l+1}_{m,n} f\left(x^{l}_{i',j'}\right)\right)$$
$$= w^{l+1}_{m,n}\frac{\partial}{\partial x^{l}_{i',j'}}\left(f\left(x^{l}_{i',j'}\right)\right)$$
$$= w^{l+1}_{m,n} f'\left(x^{l}_{i',j'}\right)$$

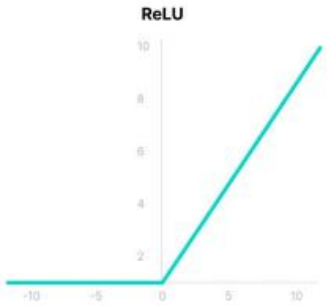
as for all except the components where m=m′ and n=n′, $f\left(x^{l}_{i'-m+m',j'-n+n'}\right)$ becomes $f\left(x^{l}_{i',j'}\right)$ and $w^{l+1}_{m',n'}$ becomes $w^{l+1}_{m,n}$.

Substituting $\frac{\partial x^{l+1}_{i'-m,j'-n}}{\partial x^{l}_{i',j'}}$ in $\frac{\partial E}{\partial x^{l}_{i',j'}}$ will give us:

$$\frac{\partial E}{\partial x^{l}_{i',j'}} = \sum_{m=0}^{k_1-1}\sum_{n=0}^{k_2-1} \delta^{l+1}_{i'-m,j'-n} w^{l+1}_{m,n} f'\left(x^{l}_{i',j'}\right)$$

Exercise 2: Comparison between ReLU and Leaky ReLU

ReLU consists of the keyword Rectified Linear Unit. If you look at the picture below, it looks like a linear function. The problem with linear functions was that their derivatives were fixed numbers, that's why we couldn't use them. But the ReLU function is derivable and we can use it for backpropagation and it has a little computational cost for us. The main point in the ReLU function is that this activation function does not activate all neurons and only activates neurons whose output is greater than one.

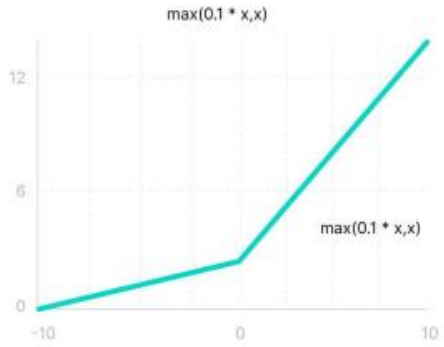


$$ReLU$$

$$f(x) = max\,(0, x)$$

Advantages: Because only a number of neurons are activated, the computational cost of this function is low. For this reason, this function is used in many cases such as text processing, image processing, and audio processing. Due to its linear feature, this function accelerates the gradient descent operation to find the global minimum value.
Disadvantages: The main drawback of this function is a problem called "Dying ReLU". Dying ReLU means that some ReLU neurons die and become inactive and the output becomes 0 for all inputs. In this case, there is no current gradient and if the number of inactive neurons in the neural network increases, the performance of the model will be affected.

Leaky ReLU has all the advantages of ReLU and also solves the problem of neuron death. With this change in the ReLU function, the gradient is no longer zero for negative values, and we can use it for cases where the output of the neuron is negative.



Leaky ReLU

$$f(x) = max\ (0.1x, x)$$

Disadvantages: Predictions may not be consistent with negative inputs. The gradient for negative numbers is a small value, which can increase the cost of learning for us.

Conclusion: Leaky ReLU increases the speed of training. There is some evidence that a "mean activation" close to 0 makes training faster. Leaky ReLU is more "balanced" and may learn faster.

Vanishing gradient: When we train the neural network using gradient-based methods, such as backpropagation, we encounter the problem of gradient vanishing. This problem makes it difficult to learn and update the weights in the initial layers of the network; In fact, the problem of gradient vanishing is one of the unstable behaviors of the network that we may encounter when training the network. When a neural network uses special activation functions, such as the Sigmoid, the number of layers increases, the value of the gradient of the loss function approaches zero, large-scale input values are placed in a small interval between zero and 1; So, when a very large change occurs in the input value of the function, the output of the function changes only a small amount; This means that the value of its derivative becomes very small. As we saw earlier ReLU and Leaky ReLU both can tackle this problem