A Tutorial on Neural Networks and Convolutional Neural Networks

Halim Noor

Introduction

Artificial Neural Network or simply neural network (ANN) is a machine learning technique that is modeled loosely after the human brain. It is a powerful, scalable and versatile technique that has been used to solve highly complex problems such as pattern recognition, machine translation and e-mail spam filtering. A neural network composes of layers of artificial neurons or nodes. Each layer's neurons are connected to adjacent or next layer's neurons. Each connection (or edge) is responsible to carry a signal from one neuron to another neuron i.e. from the input layer to the output layer. The layer(s) in between the input and output layers are called hidden layer. The connections have a weight that refers to the strength of the connection between two neurons.

Artificial Neuron

A neuron is the fundamental unit of a neural network. Figure 1 shows a neuron that accepts three inputs which may come from the external environment (inputs) or maybe from the outputs of other neurons. Each input is associated with a connection weight. The neuron computes the summation of the weighted inputs, then applies an activation function to that sum to produce an output. The output can be defined as follows.

$$z = \left[\sum_{i=0}^{2} x_i w_i\right] + b$$

$$a = f(z)$$

In addition to inputs, a neuron may have a bias unit which is always has the value of 1. We will see the function of bias with example later. A neuron is characterized by an activation function. The activation function represents the rate at which a neuron is activated or not (firing its output or not). Notice that the output of the neuron is just a summation of the input values (identity function) which essentially a linear model. A linear model is simple to solve

but limited in its ability to solve complex problems. To make a neural network that is capable of learning and solving complex tasks, a non-linear activation function is used to transform the output value. The commonly used activation functions are sigmoid, hyperbolic tangent or rectified linear unit (relu).

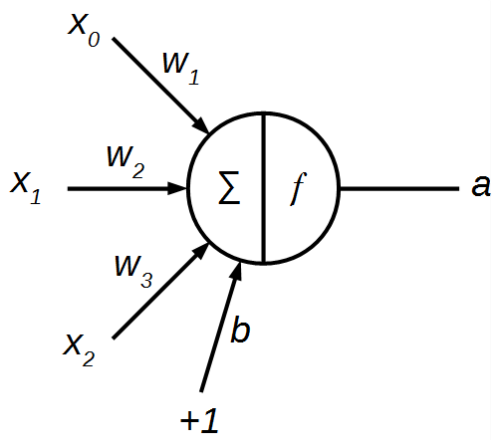

Figure 1. An artificial neuron.

For example, applying a sigmoid function to the output is defined as

$$a = f(z) = \frac{1}{1 + e^{-z}}$$

Let's plot the output of the neuron, z for input values in the range of -10 to 10 as can be seen in Figure 3.

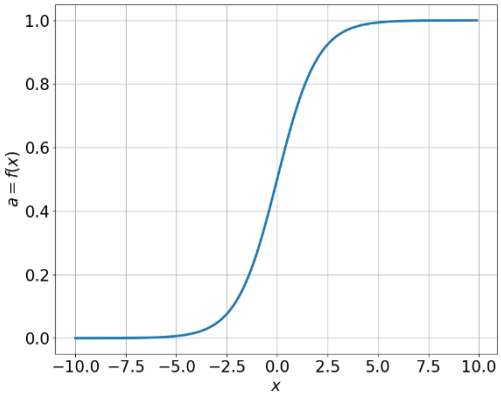

Figure 2. A sigmoid function.

As shown in Figure 2, sigmoid function has positive values for all x. Notice that the plot looks like a step function with smooth or soft edges. That's means, there is a non-zero derivative (gradient) of the function for every value of x. This is important for training the neural network which will be discussed later.

Let's see the outputs of the sigmoid function with different weight values. We define three weight values: 0.5, 1.0 and 1.5.

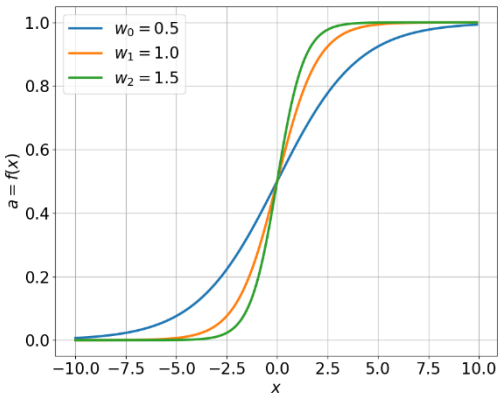

Figure 3. The output of sigmoid function with three different weight values.

As shown in Figure 3, we can see that changing the weights changes the slope of the output of the sigmoid function. The slope of the output is getting steeper as the weight is increased.

Now, to see the function of bias, we plot the output of the sigmoid function with different values of bias. The weight is equals 1.

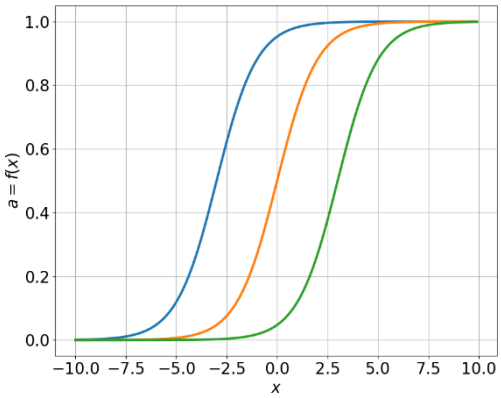

Figure 4. The output of sigmoid function with different values of bias.

As can be seen in Figure 4, changing the value of bias will shift the activation function forward or backward. That's means the bias allows the network to model a conditional relationship such as

```
if (x > a) then
   1 (the neuron is activated)
else
   0 (the neuron is not activated)
```

Activation Functions

Different types of activation functions can be used to introduce non-linear properties to a neural network. Various activation functions have been used

but the most commonly used are sigmoid (or logistic), hyperbolic tangent (tanh), and rectified linear unit (ReLU).

Sigmoid function maps the input to any value range between (0,1). It has a nice property where it tends to bring the activations towards 0 or 1 due to the steep slope in the middle of the function. However, the gradient at either tail of 0 or 1 is very small or almost zero. During backpropagation, the gradients are backpropagated (multiplied with the gradients of the neurons) through the hidden layers to adjust the weights. If the gradient of a neuron is very small, the backpropagation signal will not get through the neuron and as a result the network will barely learn.

Tanh is similar to sigmoid function. Instead of mapping the input to any value in the range of (0,1), the output of the function is in the range of $(-1,1)$. It is closely related to sigmoid because the following holds.

$$f(z) = 2\sigma(2z) - 1$$

where $\sigma(z)$ is the sigmoid function. Although they have similar properties, tanh is preferred because it often converge faster than the sigmoid [1].

ReLU is a non-linear activation function that is defined as follows.

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

In other word, ReLU is linear for all positive values and zero for all negative values as shown in Figure 5. Compared to sigmoid and tanh, ReLU is simpler to compute and therefore take less time to train and execute. Unlike sigmoid and tanh, ReLU does not suffer from saturation or vanishing gradient. A variant of ReLU called Leaky ReLU is proposed to fix dying ReLU [2]. Dying ReLU refers to a neuron that never activates due to zero gradient. Another variant of ReLU called exponential liner unit (ELU). It has been shown that ELU could speed up training and achieve higher classification accuracy [3].

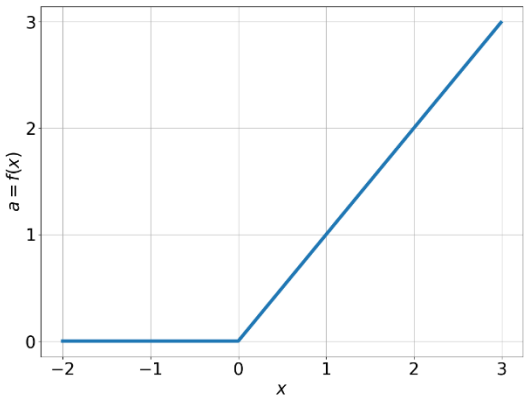

Figure 5. Rectified linear unit activation function.

Artificial Neural Network Model

An ANN is an interconnection of the artificial neurons organized in layers. The layers are called input, hidden and output layers. Feed-forward or forward propagation is the process of propagating the signal from the input layer to the output layer through the hidden layers. The hidden layers are responsible to learn non-linear combination (hidden representations or features) of the input data. Each of the connections that connect a neuron of a layer to a neuron of the adjacent layer will have an associated weight. Figure 6 illustrates an ANN consists of input and output layers with two hidden layers.

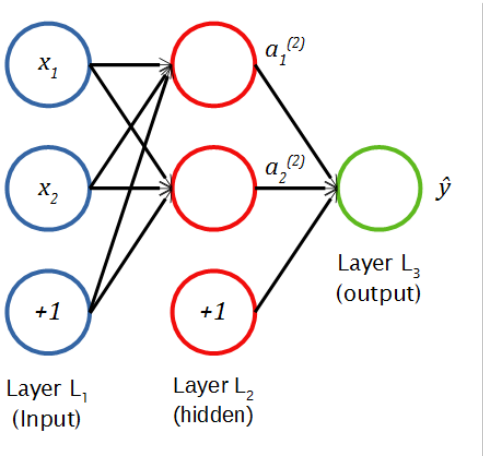

Figure 6. An ANN model.

Let K is the number of layers in a neural network. Hence, in this example $K = 3$ where L_1 is the input layer, L_2 is the hidden layer and L_K is the output layer. The weights and bias are denoted by W and b where $W_{ij}^{(l)}$ denote the weight associated with the connection between neuron j in layer l and neuron i in layer $l+1$. The $b_i^{(l)}$ is the bias associated with neuron i in layer $l+1$. We denote the j-th input as x_j. The propagation of input x_j to the neurons of the next layer is given as follows.

$$z_1^{(2)} = W_{11}^{(1)} x_1 + W_{12}^{(1)} x_2 + b_1^{(1)}$$

$$z_2^{(2)} = W_{21}^{(1)} x_1 + W_{22}^{(1)} x_2 + b_2^{(1)}$$

$$a_1^{(2)} = f\left(z_1^{(2)}\right)$$

$$a_2^{(2)} = f\left(z_2^{(2)}\right)$$

where $z_i^{(l)}$ is the summation of the weighted inputs to neuron i in layer l including the bias term. We also let $a_i^{(l)}$ denote the activation (the output of the activation function) of neuron i in layer l. The propagation of the signals to the output layer follow the same computation.

$$z_1^{(3)} = W_{11}^{(2)} a_1^{(2)} + W_{12}^{(2)} a_2^{(2)} + b^{(2)}$$

$$\hat{y}_1 = f\left(z_1^{(3)}\right)$$

Backpropagation Algorithm

A neural network needs to be trained to solve a prediction problem. The training (or learning) is a process of finding the weight and bias values that will produce the desired output at the output layer when a certain input is given to the network. How does a neural network learn? For a human to learn to do things, we will be told that we are doing it right or wrong. For example, if we are to learn to play football, we watch how the ball flies as we kick the ball. The next time we kick the ball, we remember the mistake that we have done before and improvise the movement to kick the ball a bit better. Similarly, a neural network learns to solve a problem by comparing the output that is produced by the network with the output that was meant to produce. The difference between the network's output and the true output is the amount of error that the network needs to reduce to improve its output. To reduce the error, the weights of the connections between neurons are adjusted in such a way that the output that will be produced by the network is closed to the actual output. This involve propagating the error from the output layer through the hidden layer to the input layer. Hence the process is called backpropagation.

Gradient descent is a commonly used backpropagation algorithm to adjust the weights of neurons by calculating the gradient (rate of change of error) of the cost function. Let the output of the neural network (prediction) in Figure 5 is denoted by $\hat{y}$. We define the error or the cost function with respect to a single example as follows.

$$E = J(\mathbb{W}, \mathbb{b}) = \frac{1}{2}(\hat{y} - y)^2$$

The gradient of E is obtained by calculating the partial derivatives with respect to each weight using the chain rule.

$$\frac{\partial E}{\partial W_{ij}^{(l)}} = \frac{\partial E}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z_i^{(l+1)}} \cdot \frac{\partial z_i^{(l+1)}}{\partial W_{ij}^{(l)}}$$

The derivative of E is

$$\frac{\partial E}{\partial \hat{y}} = (\hat{y} - y)$$

The derivative of sigmoid function, $f\left(z_i^{(l)}\right)$ is

$$\frac{\partial \hat{y}}{\partial z_i^{(l)}} = f\left(z_i^{(l)}\right)\left(1 - f\left(z_i^{(l)}\right)\right)$$

If W_{ij} is the weight of the connection between the neuron in L_3 (output) and the neuron in the preceding layer L_2, the derivative of the last factor is

$$\frac{\partial z_i^{(3)}}{\partial W_{ij}^{(2)}} = \frac{\partial}{\partial W_{ij}^{(2)}}\left(\sum_{j=1}^{N} W_{ij}^{(2)} a_j\right) = a_j$$

Note that only one term in $z_i^{(l)}$ depends on $W_{ij}^{(l-1)}$. If layer l is the input layer, a_j is just x_j.

To find the derivative of E with respect to $W_{ij}^{(1)}$

$$\frac{\partial E}{\partial W_{ij}^{(1)}} = \frac{\partial E}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z_i^{(3)}} \cdot \frac{\partial z_i^{(3)}}{\partial W_{ij}^{(1)}}$$

where

$$\frac{\partial z_i^{(3)}}{\partial W_{ij}^{(1)}} = \frac{\partial z_i^{(3)}}{\partial a_i^{(2)}} \cdot \frac{\partial a_i^{(2)}}{\partial z_i^{(2)}} \cdot \frac{\partial z_i^{(2)}}{\partial W_{ij}^{(1)}}$$

Let us see a numerical example of backpropagation algorithm. Assuming the weights are randomly initialized as follows.

$$W^{(1)} = \begin{bmatrix} 0.25 & 0.40 \\ 0.30 & 0.15 \end{bmatrix}$$

$$b^{(1)} = [0.20 \quad 0.15]$$

$$W^{(2)} = [0.10 \quad 0.30]$$

$$b^{(2)} = [0.20]$$

The input and output are given as follows.

$$x = [0.05 \quad 0.10]$$

$$y = [0.99]$$

and the learning rate, α equals 0.1.

Forward Pass: The output of the hidden layer, a and the prediction, $\hat{y}$ are calculated as follows.

$$z_1^{(2)} = W_{11}^{(1)} x_1 + W_{12}^{(1)} x_2 + b_1^{(1)} = 0.25 \cdot 0.05 + 0.40 \times 0.10 + 0.20 = 0.2525$$

$$a_1^{(2)} = f\left(z_1^{(2)}\right) = \frac{1}{1 + e^{-0.253}} = 0.5628$$

$$z_2^{(2)} = W_{21}^{(1)} x_1 + W_{22}^{(1)} x_2 + b_2^{(1)} = 0.30 \cdot 0.05 + 0.15 \times 0.10 + 0.15 = 0.18$$

$$a_2^{(2)} = f\left(z_2^{(2)}\right) = 0.5449$$

$$z_1^{(3)} = W_{11}^{(2)} a_1^{(2)} + W_{12}^{(2)} a_2^{(2)} + b_1^{(2)} = 0.10 \times 0.5628 + 0.30 \times 0.5449 + 0.20 = 0.4197$$

$$\hat{y} = f\left(z_1^{(3)}\right) = 0.6034$$

Notice that the output is 0.999 while the prediction of the network is 0.6034. Calculating the error using the cost function

$$E = \frac{1}{2}(0.6034 - 0.999)^2 = 0.0747$$

Backward Pass: We want to minimize the error by updating each weight in the network such that its prediction close to the output. To update $W_{11}^{(2)}$, we need to calculate its partial derivative as follows

$$\frac{\partial E}{\partial W_{11}^{(2)}} = \frac{\partial E}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z_1^{(3)}} \cdot \frac{\partial z_1^{(3)}}{\partial W_{11}^{(2)}}$$

where

$$\frac{\partial E}{\partial \hat{y}} = (\hat{y} - y) = (0.6034 - 0.999) = -0.3866$$

$$\frac{\partial \hat{y}}{\partial z_1^{(3)}} = f\left(z_1^{(3)}\right)\left(1 - f\left(z_1^{(3)}\right)\right) = \hat{y}(1 - \hat{y}) = 0.6034(1 - 0.6034) = 0.2393$$

$$\frac{\partial z_1^{(3)}}{\partial W_{11}^{(2)}} = \frac{\partial}{\partial W_{11}^{(2)}}\left(W_{11}^{(2)} a_1^{(2)} + W_{12}^{(2)} a_2^{(2)} + b_1^{(2)}\right) = a_1 = 0.5628$$

Therefore

$$\frac{\partial E}{\partial W_{11}^{(2)}} = -0.3866 \times 0.2392 \times 0.5628 = -0.0521$$

We can repeat this process to calculate $\frac{\partial E}{\partial W_{12}^{(2)}}$.

To update the inner layer's weights $W_{11}^{(1)}$, $W_{12}^{(1)}$, $W_{21}^{(1)}$ and $W_{22}^{(1)}$. We repeat the same process but slightly different since we need to see how the error caused by the neuron (that the weight $W_{11}^{(1)}$ is connected to) has propagated. To calculate the partial derivative $\frac{\partial E}{\partial W_{11}^{(1)}}$

$$\frac{\partial E}{\partial W_{11}^{(1)}} = \frac{\partial E}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z_1^{(3)}} \cdot \frac{\partial z_1^{(3)}}{\partial W_{11}^{(1)}}$$

We have calculated the values of $\frac{\partial E}{\partial \hat{y}}$ and $\frac{\partial \hat{y}}{\partial z_1^{(3)}}$ earlier.

$$\frac{\partial E}{\partial \hat{y}} = -0.3866$$

$$\frac{\partial \hat{y}}{\partial z_1^{(3)}} = 0.2393$$

Now, we need to calculate $\frac{\partial z_1^{(3)}}{\partial W_{11}^{(1)}}$. We need to see how the error caused by the neuron (that the weight $W_{11}^{(1)}$ is connected to) has propagated. Using chain rule, we can calculate the term as follows.

$$\frac{\partial z_1^{(3)}}{\partial W_{11}^{(1)}} = \frac{\partial z_1^{(3)}}{\partial a_1^{(2)}} \cdot \frac{\partial a_1^{(2)}}{\partial z_1^{(2)}} \cdot \frac{\partial z_1^{(2)}}{\partial W_{11}^{(1)}}$$

$$\frac{\partial z_1^{(3)}}{\partial a_1^{(2)}} = 0.10$$

$$\frac{\partial a_1^{(2)}}{\partial z_1^{(2)}} = a_1^{(2)}\left(1 - a_1^{(2)}\right) = 0.5628 \times (1 - 0.5628) = 0.2461$$

$$\frac{\partial z_1^{(2)}}{\partial W_{11}^{(1)}} = \frac{\partial}{\partial W_{11}^{(1)}}\left(W_{11}^{(1)} x_1 + W_{12}^{(1)} x_2 + b_1^{(1)}\right) = x_1 = 0.05$$

Therefore,

$$\frac{\partial z_1^{(3)}}{\partial W_{11}^{(1)}} = 0.10 \times 0.2461 \times 0.05 = 0.0012$$

The partial derivative $\frac{\partial E}{\partial W_{11}^{(1)}}$ is

$$\frac{\partial E}{\partial W_{11}^{(1)}} = -0.3866 \times 0.2393 \times 0.0012 = -0.00011$$

The same process is repeated for $W_{12}^{(1)}$, $W_{21}^{(1)}$ and $W_{22}^{(1)}$. Once we have calculated the gradient of E, we update the weights as follows.

$$W_{11}^{(2)} = W_{11}^{(2)} - \alpha \frac{\partial E}{\partial W_{11}^{(2)}} = 0.25 - 0.1 \times -0.0521 = 0.2552$$

$$W_{11}^{(1)} = W_{11}^{(1)} - \alpha \frac{\partial E}{\partial W_{11}^{(1)}} = 0.05 - 0.1 \times -0.00011 = 0.050011$$

Convolutional Layers

The interconnection of the neurons in the hidden layers learns the non-linear combinations or features of the input data. This is particularly useful to extract features from an image where each pixel is not individually informative. A pixel cannot tell us much about the image; it is the combination or the topology of the pixels that describe the image. This makes images different than other data such as number of rooms feature for house price prediction where the feature is informative on its own. A regular neural network does not scale well to full images. Using a neural network on images of size 28×28 would require 784 weights if the first hidden layer of the network is a single fully connected neuron. Increasing the number of neurons to two will double the number of weights to 1568. That number of weights seems manageable. However, it is clear that this fully-connected structure does not scale well to larger images. For example, images of size 300×300 would require 90000 weights and this number would increase to 270000 if the input are color images. Therefore, neural networks are not a viable option for processing high dimensional inputs.

Convolutional neural network (CNN) is a class of neural network that is suited for processing image and time series data. The name "convolutional neural network" refers to the layer(s) of the network that employ a mathematical operation called convolution. Typically, a CNN consists of convolutional layers, optional pooling layers (which will be discussed later) followed by fully-connected layers. The convolutional layers are the core building block of CNN that automatically extract the features from the input data. In convolutional layer, each hidden neuron is connected to a small subset of the input neurons. The weights are shared across the input data in order to reduce the number of parameters. The locally connected neurons or network is an important feature of CNN. Contiguous regions of pixels or samples carry meaningful information based on the assumption that adjacent pixels have stronger relationship two distant pixels. This configuration allows the network to identify patterns of spatial significance in the input data.

Figure 7 illustrates the convolution process of CNN. The black grid represents the input image and the convolution weights are represented by the red grid which are called filter or kernel. Notice that the weights are shared across the input image. The filters have relatively much smaller dimension (height and width) than the input dimension but extends through the depth of the input volume. That's means for an image with depth of 3 which is the color channels, the CNN might have a filter size of $5 \times 5 \times 3$ (5 pixels width, 5 pixels height and depth of 3) on the convolutional layer. The filter is convolved with the input image and at each position the convolution operation is computed between the input (pixel) values and the weights of the filter. The output of the convolution layer is represented by the blue grid (the hidden neurons). The output is called feature map, represents the features at every spatial region of the input image.

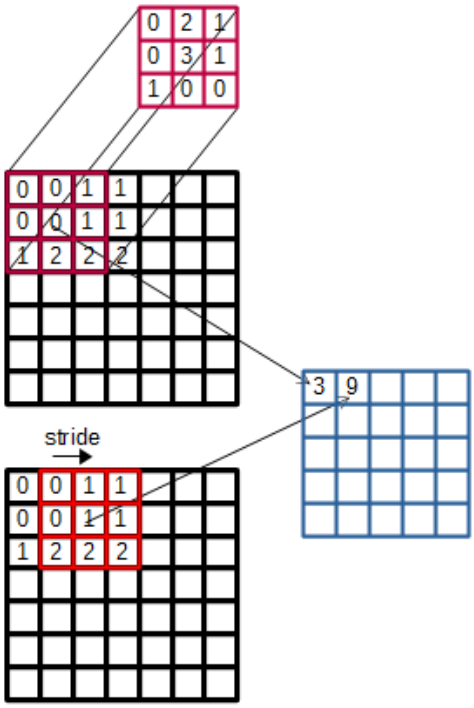

Figure 7. The process of convolution of CNN.

Hyperparameters: Stride, Padding and Depth

The size and volume of the feature map can be controlled with three hyperparameters: stride, zero-padding and depth.

Stride: The stride defines the number of pixels that the filter will be shifted. When the stride is 1 then the filters will be moved one pixel at a time. If the stride is 2 then the filters will be moved two pixels at a time. As a result, the convolutional layer will produce a smaller feature map.

Zero-padding: The size of the spatial dimensions decrease as convolutions are applied to the image. As shown in Figure 7, when a filter of 3 × 3 is applied to a 7 × 7 input image, the size of the output would be 5 × 5. The size of the image will decrease further in the following layers until it becomes too small. Moreover, The pixels at the corner will be processed only once which means we are loosing information on the corner of the image. Zero-padding allows us to pad the image with zeros around the border as shown in Figure 8. The output of the convolution will have the same spatial size as the input image.

Depth: The depth refers to the number of filters in a convolutional layer which corresponds to the number of feature maps that are produced by the convolutional layer.

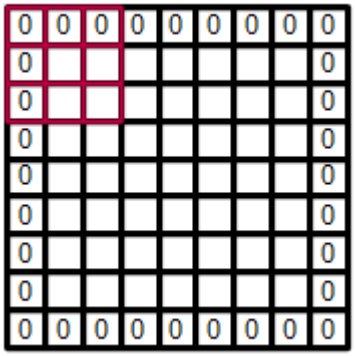

Figure 8. Applying convolution to the image with zero-padding will produce an output with the same size.

Pooling Layers

The pooling layer allows us to progressively reduce the size of the representation in order to reduce the number of parameters. Typically, a pooling layer is inserted in between two successively convolutional layers. Pooling layers have filters that partition the feature map into (non) overlapping regions and take the maximum or mean for each region. The commonly used pooling layer is MAX operation with filter of size 2 × 2 and stride equals 2 as shown in Figure 9.

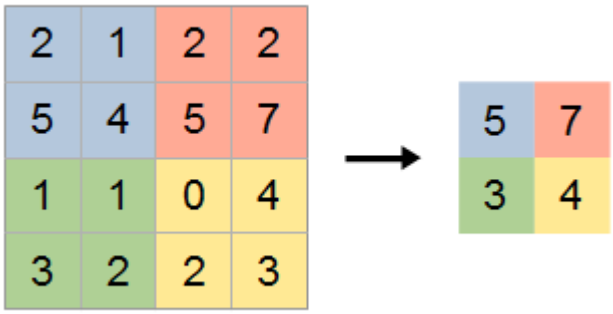

Figure 9. Max pooling with a 2 × 2 filter and stride equals 2.

Convolutional Neural Network

One of the early convolutional neural networks is called LeNet. LeNet was proposed by Yann Lecun to classify handwritten digits [4]. The size of the input image is 28 × 28 with zero-padding to increase its size to 32 × 32. The second convolutional layer does not use any padding. Each convolutional layer is followed by pooling layer to reduce the size of the feature maps. As discussed above, these layers are responsible for extracting relevant and discriminative features from the input data. There are three fully-connected layers, two hidden layers and one output layer which act as a classifier. Figure 10 and Table 1 show the architecture and parameters of the network. Numerous convolutional neural networks have been proposed for image recognition.

Some of the influential architectures are AlexNet (2012) [5], GoogLeNet (2014) [6] and DenseNet (2016) [7].

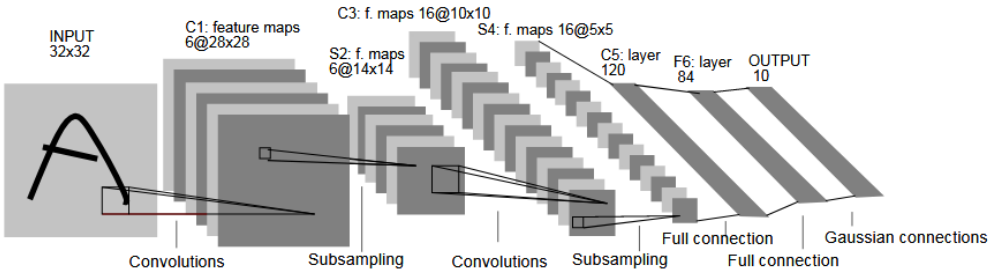

Figure 10. The architecture of LeNet.

Layer	Parameters
Convolution	5 × 5, stride 1, 6 maps, padding 2
Max pooling	2 × 2, stride 2
Convolution	5 × 5, stride 1, 16 maps
Max pooling	2 × 2, stride 2
Fully connected	120
Fully connected	84
Fully connected (output)	10

References

[1] Y. A. LeCun, L. Bottou, G. B. Orr, and K.-R. Müller, "Efficient BackProp," in *Neural Networks: Tricks of the Trade: Second Edition*, G. Montavon, G. B. Orr, and K.-R. Müller, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 9–48.

[2] A. L. Maas, "Rectifier Nonlinearities Improve Neural Network Acoustic Models," 2013.

[3] D.-A. Clevert, T. Unterthiner, and S. Hochreiter, "Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs)," *ArXiv151107289 Cs*, Nov. 2015.

[4] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proc. IEEE*, vol. 86, no. 11, pp. 2278–2324, Nov. 1998.

[5] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, 2012, pp. 1097–1105.

[6] C. Szegedy *et al.*, "Going Deeper with Convolutions," *ArXiv14094842 Cs*, Sep. 2014.

[7] G. Huang, Z. Liu, L. van der Maaten, and K. Q. Weinberger, "Densely Connected Convolutional Networks," *ArXiv160806993 Cs*, Aug. 2016.