

Demystifying the Mathematics Behind Convolutional Neural Networks (CNNs)

[ALGORITHM](#)[COMPUTER VISION](#)[DEEP LEARNING](#)[INTERMEDIATE](#)[PYTHON](#)[STRUCTURED DATA](#)[SUPERVISED](#)

Overview

- Convolutional neural networks (CNNs) are all the rage in the deep learning and computer vision community
- How does this CNN architecture work? We'll explore the math behind the building blocks of a convolutional neural network
- We will also build our own CNN from scratch using NumPy

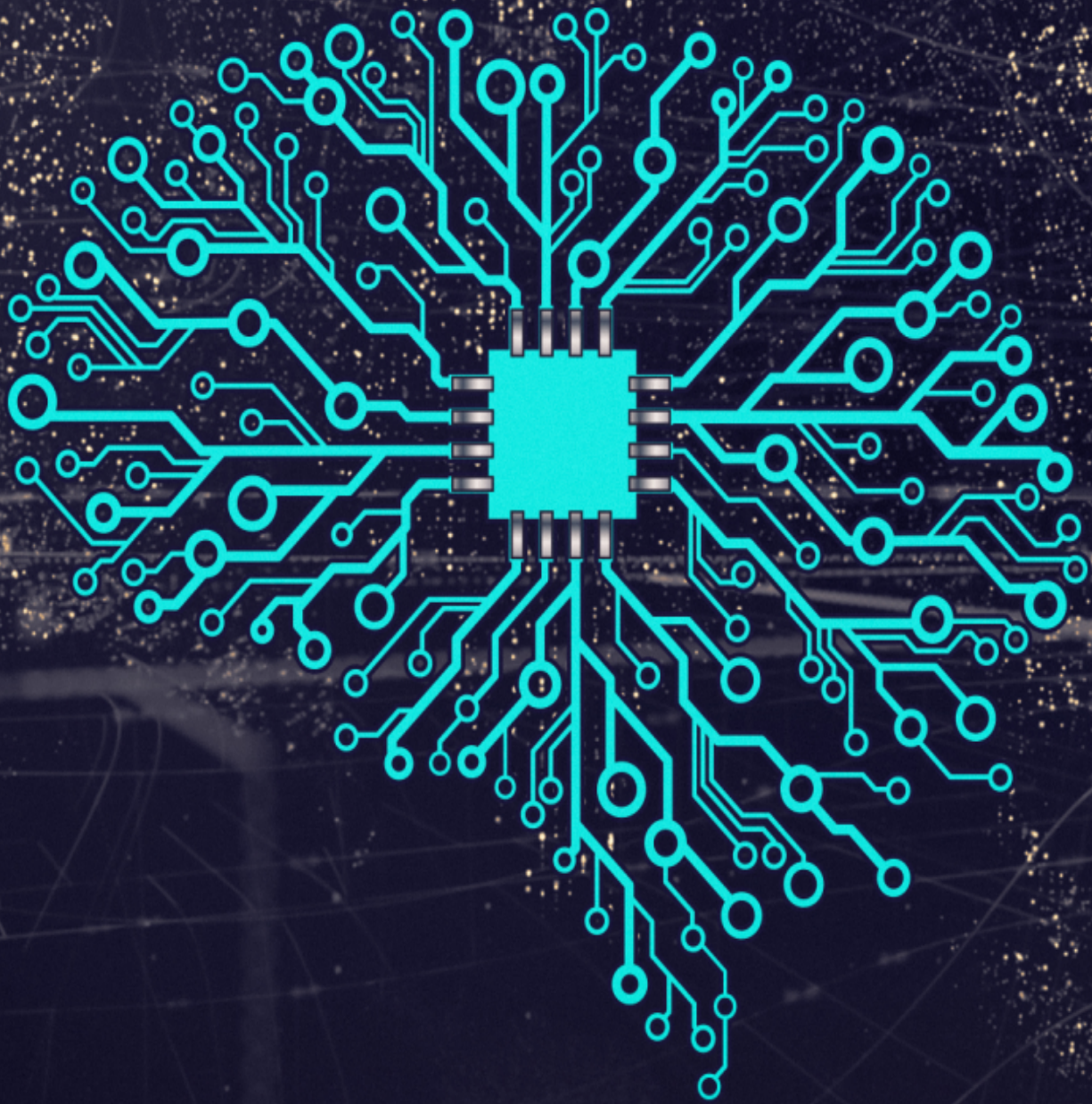
Introduction

Convolutional neural network (CNN) – almost sounds like an amalgamation of biology, art and mathematics. In a way, that's exactly what it is (and what this article will cover).

CNN-powered deep learning models are now ubiquitous and you'll find them sprinkled into various [computer vision](#) applications across the globe. Just like [XGBoost](#) and other popular machine learning algorithms, convolutional neural networks came into the public consciousness through a [hackathon](#) (the ImageNet competition in 2012).

These [neural networks](#) have caught inspiration like fire since then, expanding into various research areas. Here are just a few popular computer vision applications where CNNs are used:

- Facial recognition systems
- Analyzing and parsing through documents
- Smart cities (traffic cameras, for example)
- Recommendation systems, among other use cases



But why does a convolutional neural network work so well? How does it perform better than the traditional ANNs (Artificial neural network)? Why do deep learning experts love it?

To answer these questions, we must understand how a CNN actually works under the hood. In this article, we will go through the mathematics behind a CNN model and we'll then build our own CNN from scratch.

If you prefer a course format where we cover this content in stages, you can enrol in this free course: [Convolutional Neural Networks from Scratch](#)

Note: If you're new to neural networks, I highly recommend checking out our popular free course:

- [Introduction to Neural Networks](#)

Table of Contents

1. Introduction to Neural Networks
 1. Forward Propagation
 2. Backward Propagation

2. Convolutional Neural Network (CNN) Architecture

3. Forward Propagation in CNNs

1. Convolutional Layer
2. Linear Transformation
3. Non-linear Transformation
4. Forward Propagation Summary

4. Backward Propagation in CNNs

1. Fully Connected Layer
2. Convolution Layer

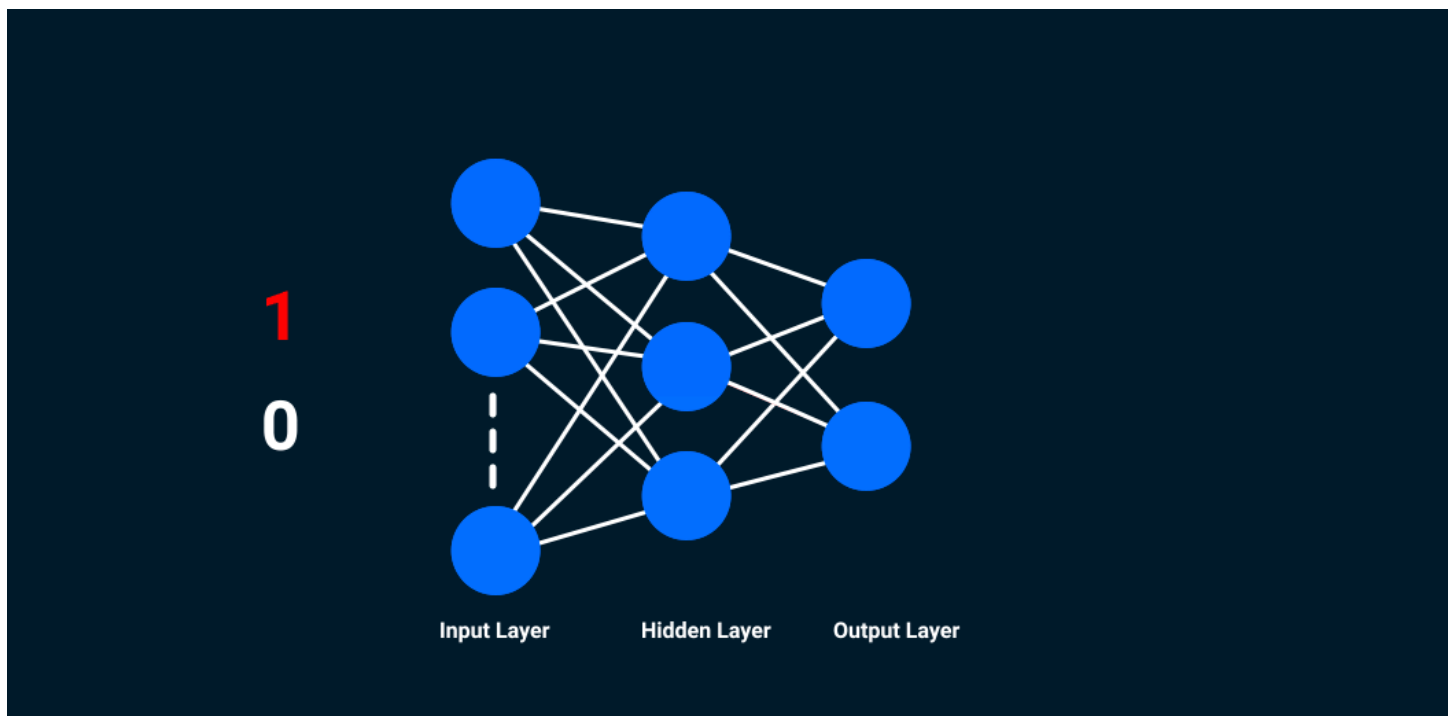
5. CNN from Scratch using NumPy

Introduction to Neural Networks

Neural Networks are at the core of all [deep learning algorithms](#). But before you deep dive into these algorithms, it's important to have a good understanding of the concept of [neural networks](#).

These neural networks try to mimic the human brain and its learning process. Like a brain takes the input, processes it and generates some output, so does the neural network.

These three actions – **receiving input, processing information, generating output** – are represented in the form of layers in a neural network – input, hidden and output. Below is a skeleton of what a neural network looks like:



These individual units in the layers are called **neurons**. The complete training process of a neural network involves two steps.

1. Forward Propagation

Images are fed into the input layer in the form of numbers. These numerical values denote the intensity of pixels in the image. The neurons in the hidden layers apply a few mathematical operations on these values (which we will discuss later in this article).

In order to perform these mathematical operations, there are certain parameter values that are randomly initialized. Post these mathematical operations at the hidden layer, the result is sent to the output layer which generates the final prediction.

2. Backward Propagation

Once the output is generated, the next step is to compare the output with the actual value. Based on the final output, and how close or far this is from the actual value (error), the values of the parameters are updated. The forward propagation process is repeated using the updated parameter values and new outputs are generated.

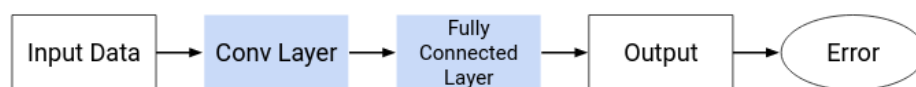
This is the base of any neural network algorithm. In this article, we will look at the forward and backward propagation steps for a convolutional neural network!

Convolutional Neural Network (CNN) Architecture

Consider this – you are asked to identify objects in two given images. How would you go about doing that? Typically, you would observe the image, try to identify different features, shapes and edges from the image. Based on the information you gather, you would say that the object is a dog or a car and so on.

This is precisely what the hidden layers in a [CNN](#) do – find features in the image. The convolutional neural network can be broken down into two parts:

- **The convolution layers:** Extracts features from the input
- **The fully connected (dense) layers:** Uses data from convolution layer to generate output



As we discussed in the previous section, there are two important processes involved in the training of any neural network:

1. **Forward Propagation:** Receive input data, process the information, and generate output
2. **Backward Propagation:** Calculate error and update the parameters of the network

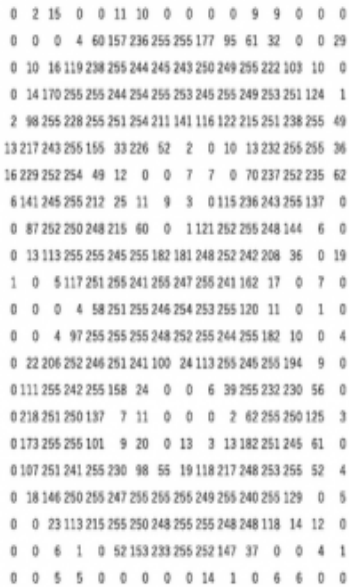
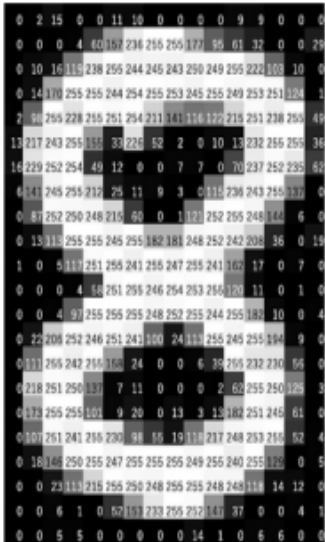
We will cover both of these one by one. Let us start with the forward propagation process.

Convolutional Neural Network (CNN): Forward Propagation

Convolution Layer

You know how we look at images and identify the object's shape and edges? A convolutional neural network does this by comparing the pixel values.

Below is an image of the number 8 and the pixel values for this image. Take a look at the image closely. You would notice that there is a significant difference between the pixel values around the edges of the number. Hence, a simple way to identify the edges is to compare the neighboring pixel value.



Do we need to traverse pixel by pixel and compare these values? No! To capture this information, the image is convolved with a filter (also known as a 'kernel').

Convolution is often represented mathematically with an asterisk * sign. If we have an input image represented as X and a filter represented with f, then the expression would be:

$$z = x * f$$

Note: To learn how filters capture information about the edges, you can go through this article:

- [Beginner-Friendly Techniques to Extract Features from Image Data](#)

Let us understand the process of convolution using a simple example. Consider that we have an image of size 3 x 3 and a filter of size 2 x 2:

1	7	2
11	1	23
2	2	2

1	1
0	1

The filter goes through the patches of images, performs an element-wise multiplication, and the values are summed up:

$$(1 \times 1 + 7 \times 1 + 11 \times 0 + 1 \times 1) = 9 \quad (7 \times 1 + 2 \times 1 + 1 \times 0 + 23 \times 1) = 32 \quad (11 \times 1 + 1 \times 1 + 2 \times 0 + 2 \times 1) = 14 \quad (1 \times 1 + 23 \times 1 + 2 \times 0 + 2 \times 1) = 26$$

1	7	2			
11	1	23	*	1	1
2	2	2		0	1
				9	

Look at that closely – you’ll notice that the filter is considering a small portion of the image at a time. We can also imagine this as a single image broken down into smaller patches, each of which is convolved with the filter.

1	7	2		
11	1	23	*	1
2	2	2		1

In the above example, we had an input of shape (3, 3) and a filter of shape (2, 2). Since the dimensions of image and filter are very small, it’s easy to interpret that the shape of the output matrix is (2, 2). But how would we find the shape of an output for more complex inputs or filter dimensions? There is a simple formula to do so:

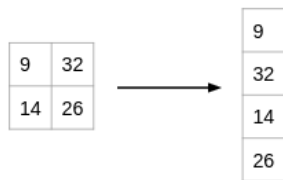
$$\text{Dimension of image} = (n, n) \quad \text{Dimension of filter} = (f, f) \quad \text{Dimension of output will be } ((n-f+1), (n-f+1))$$

You should have a good understanding of how a convolutional layer works at this point. Let us move to the next part of the [CNN](#) architecture.

Fully Connected Layer

So far, the convolution layer has extracted some valuable features from the data. These features are sent to the fully connected layer that generates the final results. The fully connected layer in a CNN is nothing but the traditional neural network!

The output from the convolution layer was a 2D matrix. Ideally, we would want each row to represent a single input image. In fact, the fully connected layer can only work with 1D data. Hence, the values generated from the previous operation are first converted into a 1D format.

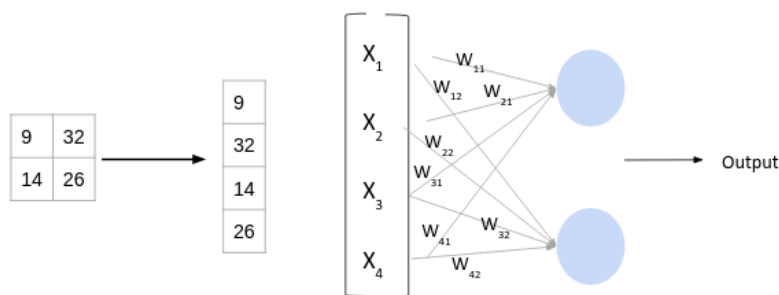


Once the data is converted into a 1D array, it is sent to the fully connected layer. All of these individual values are treated as separate features that represent the image. The fully connected layer performs two operations on the incoming data – **a linear transformation and a non-linear transformation**.

We first perform a linear transformation on this data. The equation for linear transformation is:

$$Z = W^T \cdot X + b$$

Here, X is the input, W is weight, and b (called bias) is a constant. Note that the W in this case will be a matrix of (randomly initialized) numbers. Can you guess what would be the size of this matrix?



Considering the size of the matrix is (m, n) – m will be equal to the number of features or inputs for this layer. Since we have 4 features from the convolution layer, m here would be 4. The value of n will depend on the number of neurons in the layer. For instance, if we have two neurons, then the shape of weight matrix will be (4, 2):

$$\begin{matrix} X = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} & W = \begin{bmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \\ w_{31} & w_{32} \\ w_{41} & w_{42} \end{bmatrix} & b = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} \\ \text{Input Data} & \text{Randomly Initialized} & \text{Randomly Initialized} \\ & \text{Weight Matrix} & \text{bias Matrix} \end{matrix}$$

Having defined the weight and bias matrix, let us put these in the equation for linear transformation:

$$\begin{aligned} Z &= W^T \cdot X + b \\ Z &= \begin{bmatrix} W_{11} & W_{12} & W_{13} & W_{14} \\ W_{21} & W_{22} & W_{23} & W_{24} \end{bmatrix} \begin{bmatrix} X_1 \\ X_2 \\ X_3 \\ X_4 \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} \\ Z_{2x2} &= \begin{bmatrix} W_{11}X_1 + W_{12}X_2 + W_{13}X_3 + W_{14}X_4 \\ W_{21}X_1 + W_{22}X_2 + W_{23}X_3 + W_{24}X_4 \end{bmatrix} \end{aligned}$$

Now, there is one final step in the forward propagation process – the non-linear transformations. Let us understand the concept of non-linear transformation and it's role in the forward propagation process.

Non-Linear transformation

The linear transformation alone cannot capture complex relationships. Thus, we introduce an additional component in the network which adds non-linearity to the data. This new component in the architecture is called the **activation function**.

There are a number of activation functions that you can use – here is the complete list:

- [Fundamentals of Deep Learning – Activation Functions and When to Use Them?](#)

These activation functions are added at each layer in the neural network. The activation function to be used will depend on the type of problem you are solving.

We will be working on a binary classification problem and will use the Sigmoid activation function. Let's quickly look at the mathematical expression for this:

$$f(x) = 1/(1+e^{-x})$$

The range of a Sigmoid function is between 0 and 1. This means that for any input value, the result would always be in the range (0, 1). A Sigmoid function is majorly used for binary classification problems and we will use this for both convolution and fully-connected layers.

Let's quickly summarize what we've covered so far.

Forward Propagation Summary

Step 1: Load the input images in a variable (say X)

Step 2: Define (randomly initialize) a filter matrix. Images are convolved with the filter

$$Z_1 = X * f$$

Step 3: Apply the Sigmoid activation function on the result

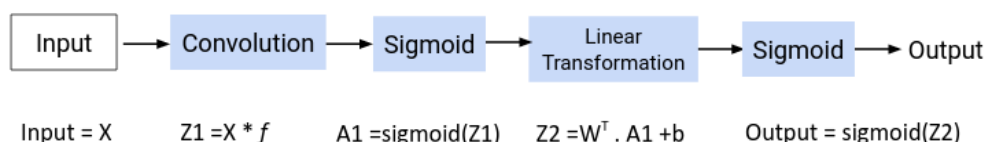
$A = \text{sigmoid}(Z_1)$

Step 4: Define (randomly initialize) weight and bias matrix. Apply linear transformation on the values

$$Z_2 = W^T \cdot A + b$$

Step 5: Apply the Sigmoid function on the data. This will be the final output

$$O = \text{sigmoid}(Z_2)$$



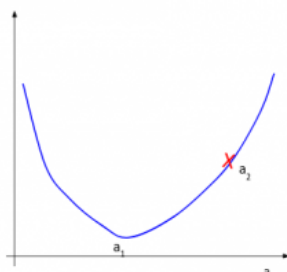
Now the question is – how are the values in the filter decided? The CNN model treats these values as parameters, which are randomly initialized and learned during the training process. We will answer this in the next section.

Convolutional Neural Network (CNN): Backward Propagation

During the forward propagation process, we randomly initialized the weights, biases and filters. These values are treated as parameters from the convolutional neural network algorithm. **In the backward propagation process, the model tries to update the parameters such that the overall predictions are more accurate.**

For updating these parameters, we use the [gradient descent technique](#). Let us understand the concept of gradient descent with a simple example.

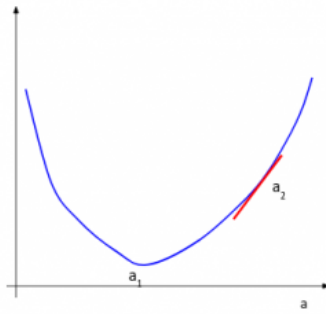
Consider that following in the curve for our loss function where we have a parameter a:



During the random initialization of the parameter, we get the value of a as a_2 . It is clear from the picture that the minimum value of loss is at a_1 and not a_2 . The gradient descent technique tries to find this value of parameter (a) at which the loss is minimum.

We understand that we need to update the value a_2 and bring it closer to a_1 . To decide the direction of movement, i.e. whether to increase or decrease the value of the parameter, we calculate the gradient or

slope at the current point.



Based on the value of the gradient, we can determine the updated parameter values. When the slope is negative, the value of the parameter will be increased, and when the slope is positive, the value of the parameter should be decreased by a small amount.

Here is a generic equation for updating the parameter values:

$$\text{new_parameter} = \text{old_parameter} - (\text{learning_rate} * \text{gradient_of_parameter})$$

The learning rate is a constant that controls the amount of change being made to the old value. The slope or the gradient determine the direction of the new values, that is, should the values be increased or decreased. So, we need to find the gradients, that is, change in error with respect to the parameters in order to update the parameter values.

If you want to read about the gradient descent technique in detail, you can go through the below article:

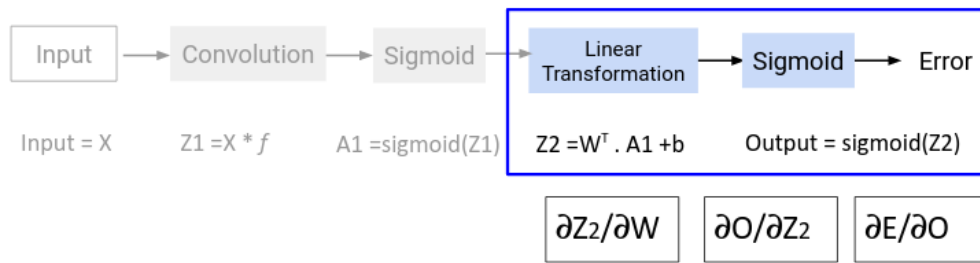
- [Introduction to Gradient Descent Algorithm](#)

We know that we have three parameters in a CNN model – weights, biases and filters. Let us calculate the gradients for these parameters one by one.

Backward Propagation: Fully Connected Layer

As discussed previously, the fully connected layer has two parameters – weight matrix and bias matrix. Let us start by calculating the change in error with respect to weights – $\partial E / \partial W$.

Since the error is not directly dependent on the weight matrix, we will use the concept of chain rule to find this value. The computation graph shown below will help us define $\partial E / \partial W$:



Backward Propagation (Fully Connected layer)

$$\frac{\partial E}{\partial W} = \frac{\partial E}{\partial O} \cdot \frac{\partial O}{\partial Z_2} \cdot \frac{\partial Z_2}{\partial W}$$

We will find the values of these derivatives separately.

1. Change in error with respect to output

Suppose the actual values for the data are denoted as y' and the predicted output is represented as O . Then the error would be given by this equation:

$$E = (y' - O)^2 / 2$$

If we differentiate the error with respect to the output, we will get the following equation:

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

2. Change in output with respect to Z_2 (linear transformation output)

To find the derivative of output O with respect to Z_2 , we must first define O in terms of Z_2 . If you look at the computation graph from the forward propagation section above, you would see that the output is simply the sigmoid of Z_2 . Thus, $\frac{\partial O}{\partial Z_2}$ is effectively the derivative of Sigmoid. Recall the equation for the Sigmoid function:

$$f(x) = 1 / (1 + e^{-x})$$

The derivative of this function comes out to be:

$$f'(x) = (1 + e^{-x})^{-1} [1 - (1 + e^{-x})^{-1}] \quad f'(x) = \text{sigmoid}(x)(1 - \text{sigmoid}(x)) \quad \frac{\partial O}{\partial Z_2} = (O)(1 - O)$$

You can read about the complete derivation of the Sigmoid function [here](#).

3. Change in Z₂ with respect to W (Weights)

The value Z₂ is the result of the linear transformation process. Here is the equation of Z₂ in terms of weights:

$$Z_2 = W^T \cdot A_1 + b$$

On differentiating Z₂ with respect to W, we will get the value A₁ itself:

$$\partial Z_2 / \partial W = A_1$$

Now that we have the individual derivations, we can use the chain rule to find the change in error with respect to weights:

$$\partial E / \partial W = \partial E / \partial O \cdot \partial O / \partial Z_2 \cdot \partial Z_2 / \partial W \quad \partial E / \partial W = -(y' - o) \cdot \text{sigmoid}' \cdot A_1$$

The shape of $\partial E / \partial W$ will be the same as the weight matrix W. We can update the values in the weight matrix using the following equation:

$$W_{\text{new}} = W_{\text{old}} - lr * \partial E / \partial W$$

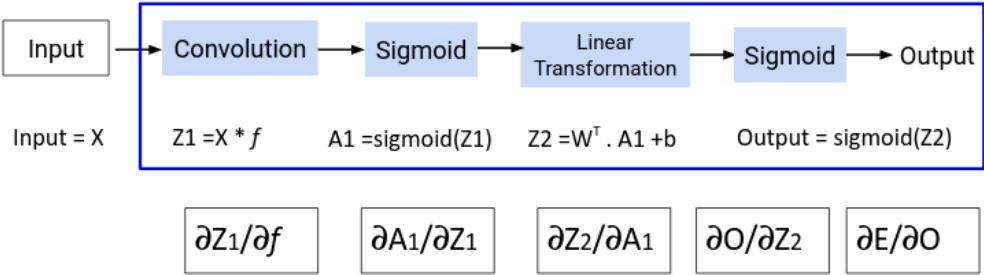
Updating the bias matrix follows the same procedure. Try to solve that yourself and share the final equations in the comments section below!

Backward Propagation: Convolution Layer

For the convolution layer, we had the filter matrix as our parameter. During the forward propagation process, we randomly initialized the filter matrix. We will now update these values using the following equation:

$$\text{new_parameter} = \text{old_parameter} - (\text{learning_rate} * \text{gradient_of_parameter})$$

To update the filter matrix, we need to find the gradient of the parameter – dE/df. Here is the computation graph for backward propagation:



Backward Propagation (Convolution layer)

From the above graph we can define the derivative $\partial E / \partial f$ as:

$$\partial E / \partial f = \partial E / \partial O . \partial O / \partial Z_2 . \partial Z_2 / \partial A_1 . \partial A_1 / \partial Z_1 . \partial Z_1 / \partial f$$

We have already determined the values for $\partial E / \partial O$ and $\partial O / \partial Z_2$. Let us find the values for the remaining derivatives.

1. Change in Z_2 with respect to A_1

To find the value for $\partial Z_2 / \partial A_1$, we need to have the equation for Z_2 in terms of A_1 :

$$Z_2 = W^T . A_1 + b$$

On differentiating the above equation with respect to A_1 , we get W^T as the result:

$$\partial Z_2 / \partial A_1 = W^T$$

2. Change in A_1 with respect to Z_1

The next value that we need to determine is $\partial A_1 / \partial Z_1$. Have a look at the equation of A_1

$$A_1 = \text{sigmoid}(Z_1)$$

This is simply the Sigmoid function. The derivative of Sigmoid would be:

$$\partial A_1 / \partial Z_1 = (A_1)(1 - A_1)$$

3. Change in Z_1 with respect to filter f

Finally, we need the value for $\partial Z_1 / \partial f$. Here's the equation for Z_1

$$Z_1 = X * f$$

Differentiating Z with respect to X will simply give us X :

$$\partial Z_1 / \partial f = X$$

Now that we have all the required values, let's find the overall change in error with respect to the filter:

$$\partial E / \partial f = \partial E / \partial O \cdot \partial O / \partial Z_2 \cdot \partial Z_2 / \partial A_1 \cdot \partial A_1 / \partial Z_1 * \partial Z_1 / \partial f$$

Notice that in the equation above, the value $(\partial E / \partial O \cdot \partial O / \partial Z_2 \cdot \partial Z_2 / \partial A_1 \cdot \partial A_1 / \partial Z_1)$ is convolved with $\partial Z_1 / \partial f$ instead of using a simple dot product. Why? The main reason is that during forward propagation, we perform a convolution operation for the images and filters.

This is repeated in the backward propagation process. Once we have the value for $\partial E / \partial f$, we will use this value to update the original filter value:

$$f = f - lr * (\partial E / \partial f)$$

This completes the backpropagation section for convolutional neural networks. It's now time to code!

CNN from Scratch using NumPy

Excited to get your hands dirty and design a convolutional neural network from scratch? The wait is over!

We will start by loading the required libraries and dataset. Here, **we will be using the MNIST dataset which is present within the *keras.datasets* library.**

For the purpose of this tutorial, we have selected only the first 200 images from the dataset. Here is the distribution of classes for the first 200 images:

```
1 # importing required libraries
2 import numpy as np
3 import pandas as pd
4 from tqdm import tqdm
5 from keras.datasets import mnist
6
7 # loading dataset
8 (x_train, y_train), (x_test, y_test) = mnist.load_data()
9
10 # selecting a subset of data (200 images)
11 x_train = x_train[:200]
12 y = y_train[:200]
13
14 X = x_train.T
15 X = X/255
16
17 y.resize((200,1))
18 y = y.T
19
20 #checking value
21 pd.Series(y[0]).value_counts()
```

loading_mnist.py hosted with ❤ by GitHub

view raw

```
1 26 9 23 7 21 4 21 3 21 0 21 2 20 6 19 8 15 5 13 dtype: int64
```

As you can see, we have ten classes here – 0 to 9. **This is a multi-class classification problem.** For now, we will start with building a simple [CNN](#) model for a binary classification problem:

```
1 # converting into binary classification
2 for i in range(y.shape[1]):
3     if y[0][i] > 4:
4         y[0][i] = 1
```



```

5         else:
6             y[0][i] = 0
7
8     #checking value counts
9     pd.Series(y[0]).value_counts()

```

[view raw](#)

binary_class.py hosted with ❤ by GitHub

```
0 109 1 91 dtype: int64
```

We will now initialize the filters for the convolution operation:

```

1 # initializing filter
2 f=np.random.uniform(size=(3,5,5))
3 f = f.T
4
5 print('Filter 1', '\n', f[:, :, 0], '\n')
6 print('Filter 2', '\n', f[:, :, 1], '\n')
7 print('Filter 3', '\n', f[:, :, 2], '\n')

```

[view raw](#)

filter_initialization.py hosted with ❤ by GitHub

Let us quickly check the shape of the loaded images, target variable and the filter matrix:

```
X.shape, y.shape, f.shape
```

```
((28, 28, 200), (1, 200), (5, 5, 3))
```

We have 200 images of dimensions (28, 28) each. For each of these 200 images, we have one class specified and hence the shape of y is (1, 200). Finally, we defined 3 filters, each of dimensions (5, 5).

The next step is to prepare the data for the convolution operation. As we discussed in the forward propagation section, the image-filter convolution can be treated as a single image being divided into multiple patches:

$$\begin{array}{|c|c|c|} \hline 1 & 7 & 2 \\ \hline 11 & 1 & 23 \\ \hline 2 & 2 & 2 \\ \hline \end{array} * \begin{array}{|c|c|} \hline 1 & 1 \\ \hline 0 & 1 \\ \hline \end{array}$$

For every single image in the data, we will create smaller patches of the same dimension as the filter matrix, which is (5, 5). Here is the code to perform this task:

```

1 # Generating patches from images
2 new_image = []
3
4 # for number of images

```

```

5     for k in range(X.shape[2]):
6         # sliding in horizontal direction
7         for i in range(X.shape[0]-f.shape[0]+1):
8             # sliding in vertical direction
9             for j in range(X.shape[1]-f.shape[1]+1):
10                 new_image.append(X[:, :, k][i:i+f.shape[0], j:j+f.shape[1]])
11
12 # resizing the generated patches as per number of images
13 new_image = np.array(new_image)
14 new_image.resize((X.shape[2], int(new_image.shape[0]/X.shape[2]), new_image.shape[1], new_image.shape[2]))
15 new_image.shape

```

create_patches.py hosted with ❤ by GitHub

[view raw](#)

(200, 576, 5, 5)

We have everything we need for the forward propagation process of the convolution layer. Moving on to the next section for forward propagation, we need to initialize the weight matrix for the fully connected layer. The size of the weight matrix will be (m, n) – where m is the number of features as input and n will be the number of neurons in the layer.

What would be the number of features? We know that we can determine the shape of the output image using this formula:

$((n-f+1), (n-f+1))$

Since the fully connected layer only takes 1D input, we will flatten this matrix. This means the number of features or input values will be:

$((n-f+1) \times (n-f+1) \times \text{num_of_filter})$

Let us initialize the weight matrix:

```

1 # number of features in data set
2 s_row = X.shape[0] - f.shape[0] + 1
3 s_col = X.shape[1] - f.shape[1] + 1
4 num_filter = f.shape[2]
5
6 inputlayer_neurons = (s_row)*(s_col)*(num_filter)
7 output_neurons = 1
8
9 # initializing weight
10 wo=np.random.uniform(size=(inputlayer_neurons,output_neurons))

```

weight.py hosted with ❤ by GitHub

[view raw](#)

Finally, we will write the code for the activation function which we will be using for the convolution neural network architecture.

Now, we have already converted the original problem into a binary classification problem. Hence, we will be using Sigmoid as our activation function. We have already discussed the mathematical equation for Sigmoid and its derivative. Here is the python code for the same:

```

1 # defining the Sigmoid Function
2 def sigmoid (x):
3     return 1/(1 + np.exp(-x))
4
5 # derivative of Sigmoid Function
6 def derivatives_sigmoid(x):
7     return x * (1 - x)

```

sigmoid.py hosted with ❤ by GitHub

[view raw](#)

Great! We have all the elements we need for the forward propagation process. Let us put these code blocks together.

First, we perform the convolution operation on the patches created. After the convolution, the results are stored in the form of a list, which is converted into an array of dimension (200, 3, 576). Here

- 200 is the number of images
- 576 is the number of patches created, and
- 3 is the number of filters we used

After the convolution operation, we apply the Sigmoid activation function:

```
1 # generating output of convolution layer
2 filter_output = []
3 # for each image
4 for i in range(len(new_image)):
5     # apply each filter
6     for k in range(f.shape[2]):
7         # do element wise multiplication
8         for j in range(new_image.shape[1]):
9             filter_output.append((new_image[i][j]*f[:, :, k]).sum())
10
11 filter_output = np.resize(np.array(filter_output), (len(new_image), f.shape[2], new_image.shape[1]))
12
13 # applying activation over convolution output
14 filter_output_sigmoid = sigmoid(filter_output)
15
16 filter_output.shape, filter_output_sigmoid.shape
```

convolution.py hosted with ❤ by GitHub

[view raw](#)

((200, 3, 576), (200, 3, 576))

After the convolution layer, we have the fully connected layer. We know that the fully connected layer will only have 1D inputs. So, we first flatten the results from the previous layer using the *reshape* function. Then, we apply the linear transformation and activation function on this data:

```
1 # generating input for fully connected layer
2 filter_output_sigmoid = filter_output_sigmoid.reshape((filter_output_sigmoid.shape[0], filter_output_sigmoid.shape[1]*filter_output_sigmoid.shape[2]))
3
4 filter_output_sigmoid = filter_output_sigmoid.T
5
6 # Linear transformation for fully Connected Layer
7 output_layer_input = np.dot(w0.T, filter_output_sigmoid)
8 output_layer_input = (output_layer_input - np.average(output_layer_input))/np.std(output_layer_input)
9
10 # activation function
11 output = sigmoid(output_layer_input)
```

linear.py hosted with ❤ by GitHub

[view raw](#)

It's time to start the code for backward propagation. Let's define the individual derivatives for backward propagation of the fully connected layer. Here are the equations we need:

$$E = (y' - 0)^2/2 \quad \partial E/\partial 0 = -(y' - 0) \quad \partial 0/\partial Z_2 = (0)(1-0) \quad \partial Z_2/\partial W = A_1$$

Let us code this in Python:

```
1 #Error
2 error = np.square(y-output)/2
3
4 #Error w.r.t Output (Gradient)
5 error_wrt_output = -(y-output)
```

```

6
7 #Error w.r.t sigmoid transformation (output_layer_input)
8 output_wrt_output_layer_input=output*(1-output)
9
10 #Error w.r.t weight
11 output_wrt_w=filter_output_sigmoid

```

[view raw](#)

derivative_fc.py hosted with ❤ by GitHub

We have the individual derivatives from the previous code block. We can now find the overall change in error w.r.t. weight using the chain rule. Finally, we will use this gradient value to update the original weight matrix:

$$W_{\text{new}} = W_{\text{old}} - lr * \partial E / \partial W$$

```

1 #delta change in w for fully connected layer
2 delta_error_fcp = np.dot(output_wrt_w, (error_wrt_output * output_wrt_output_layer_input).T)
3
4 wo = wo - lr*delta_error_fcp

```

[view raw](#)

weight_update.py hosted with ❤ by GitHub

So far we have covered backpropagation for the fully connected layer. This covers updating the weight matrix. Next, we will look at the derivatives for backpropagation for the convolutional layer and update the filters:

$$\partial E / \partial f = \partial E / \partial 0. \partial 0 / \partial Z_2. \partial Z_2 / \partial A_1. \partial A_1 / \partial Z_1 * \partial Z_1 / \partial f \quad \partial E / \partial 0 = -(y' - 0) \quad \partial 0 / \partial Z_2 = (0)(1-0) \quad \partial Z_2 / \partial A_1 = W^T \quad \partial A_1 / \partial Z_1 = A_1(1-A_1) \quad \partial Z_1 / \partial f = X$$

We will code the first four equations in Python and calculate the derivative using the *np.dot* function. Post that we need to perform a convolution operation using $\partial Z_1 / \partial f$:

```

1 #Error w.r.t sigmoid output
2 output_layer_input_wrt_filter_output_sigmoid = wo.T
3
4 #Error w.r.t sigmoid transformation
5 filter_output_sigmoid_wrt_filter_output = filter_output_sigmoid * (1-filter_output_sigmoid)
6
7 # calculating derivatives for backprop convolution
8 error_wrt_filter_output = np.dot(output_layer_input_wrt_filter_output_sigmoid.T, error_wrt_output*output_wrt_output_layer_input)
9 error_wrt_filter_output = np.average(error_wrt_filter_output, axis=1)
10 error_wrt_filter_output = np.resize(error_wrt_filter_output, (X.shape[0]-f.shape[0]+1, X.shape[1]-f.shape[1]+1, f.shape[2]))

```

[view raw](#)

derivative_conv.py hosted with ❤ by GitHub

```

1 filter_update = []
2 for i in range(f.shape[2]):
3     for j in range(f.shape[0]):
4         for k in range(f.shape[1]):
5             temp = 0
6             spos_row = j
7             spos_col = k
8             epos_row = spos_row + s_row
9             epos_col = spos_col + s_col
10            for l in range(X.shape[2]):
11                temp = temp + (X[spos_row:epos_row, spos_col:epos_col, l]*error_wrt_filter_output[:, :, i]).sum()
12            filter_update.append(temp/X.shape[2])
13
14 filter_update_array = np.array(filter_update)
15 filter_update_array = np.resize(filter_update_array, (f.shape[2], f.shape[0], f.shape[1]))

```

[view raw](#)

backprop_convolve.py hosted with ❤ by GitHub

We now have the gradient value. Let us use it to update the error:

```
1 for i in range(f.shape[2]):  
2     f[:, :, i] = f[:, :, i] - lr*filter_update_array[i]
```

[view raw](#)

filter_update.py hosted with ❤ by GitHub

End Notes

Take a deep breath – that was a lot of learning in one tutorial! Convolutional neural networks can appear to be slightly complex when you're starting out but once you get the hang of how they work, you'll feel ultra confident in yourself.

I had a lot of fun writing about what goes on under the hood of these CNN models we see everywhere these days. We covered the mathematics behind these CNN models and learned how to implement them from scratch, using just the NumPy library.

I want you to explore other concepts related to CNN, such as the padding and pooling techniques. Implement them in the code as well and share your ideas in the comments section below.

You should also try your hand at the below hackathons to practice what you've learned:

- [Practice Problem: Identify the Apparels](#)
- [Practice Problem: Identify the Digits](#)

Article Url - <https://www.analyticsvidhya.com/blog/2020/02/mathematics-behind-convolutional-neural-network/>



[Aishwarya Singh](#)

An avid reader and blogger who loves exploring the endless world of data science and artificial intelligence. Fascinated by the limitless applications of ML and AI; eager to learn and discover the depths of data science.