

# Build Your Own Convolution Neural Network in 5 mins

An introduction to CNN and code (Keras)

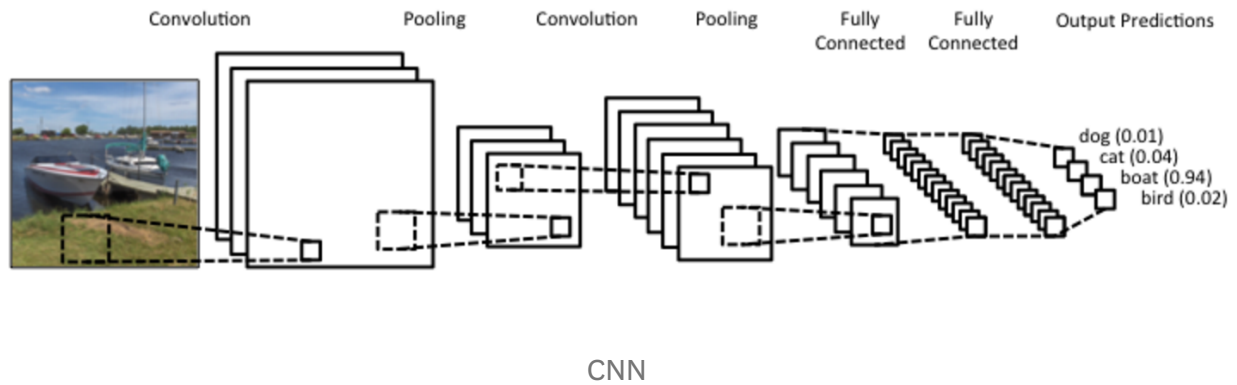


Rohith Gandhi

May 18, 2018 · 5 min read



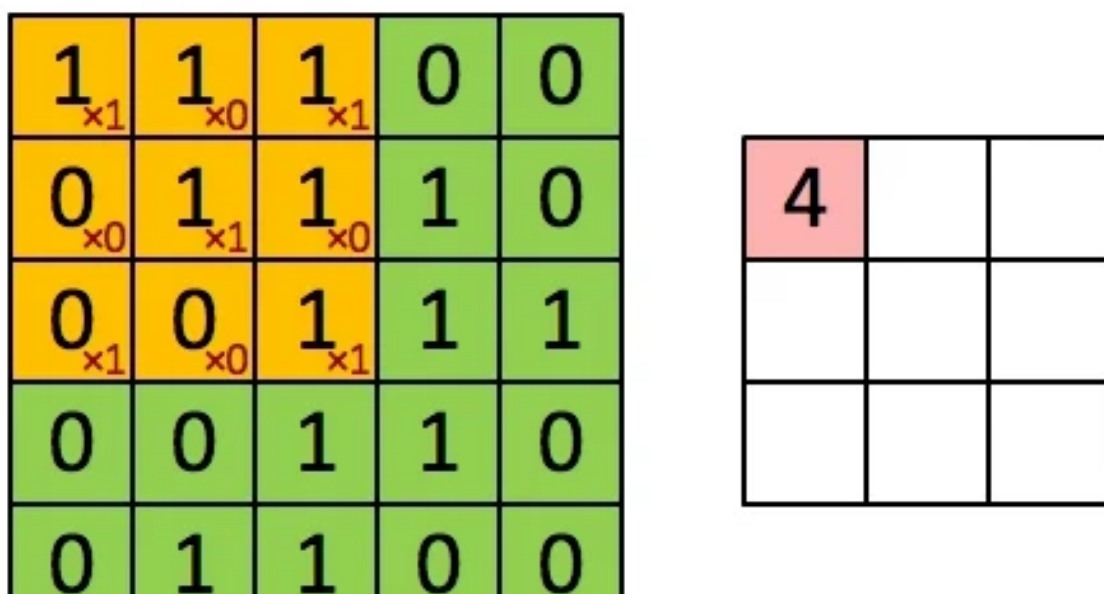
## What is a Convolutional Neural Network??



Before answering what a convolutional neural network is, I believe you guys are aware of what neural networks are. If you are shaky on the basics, check out this link. Moving on. A convolution neural network is similar to a multi-layer perceptron network. The major differences are what the network learns, how they are structured and what purpose they are mostly used for. Convolutional neural networks were also inspired from biological processes, their structure has a semblance of the visual cortex present in an animal. CNNs are largely applied in the domain of computer vision and has been highly successful in achieving state of the art performance on various test cases.

## What do the hidden layers learn??

The hidden layers in a CNN are generally convolution and pooling(downsampling) layers. In each convolution layer, we take a filter of a small size and move that filter across the image and perform convolution operations. Convolution operations are nothing but element-wise matrix multiplication between the filter values and the pixels in the image and the resultant values are summed.





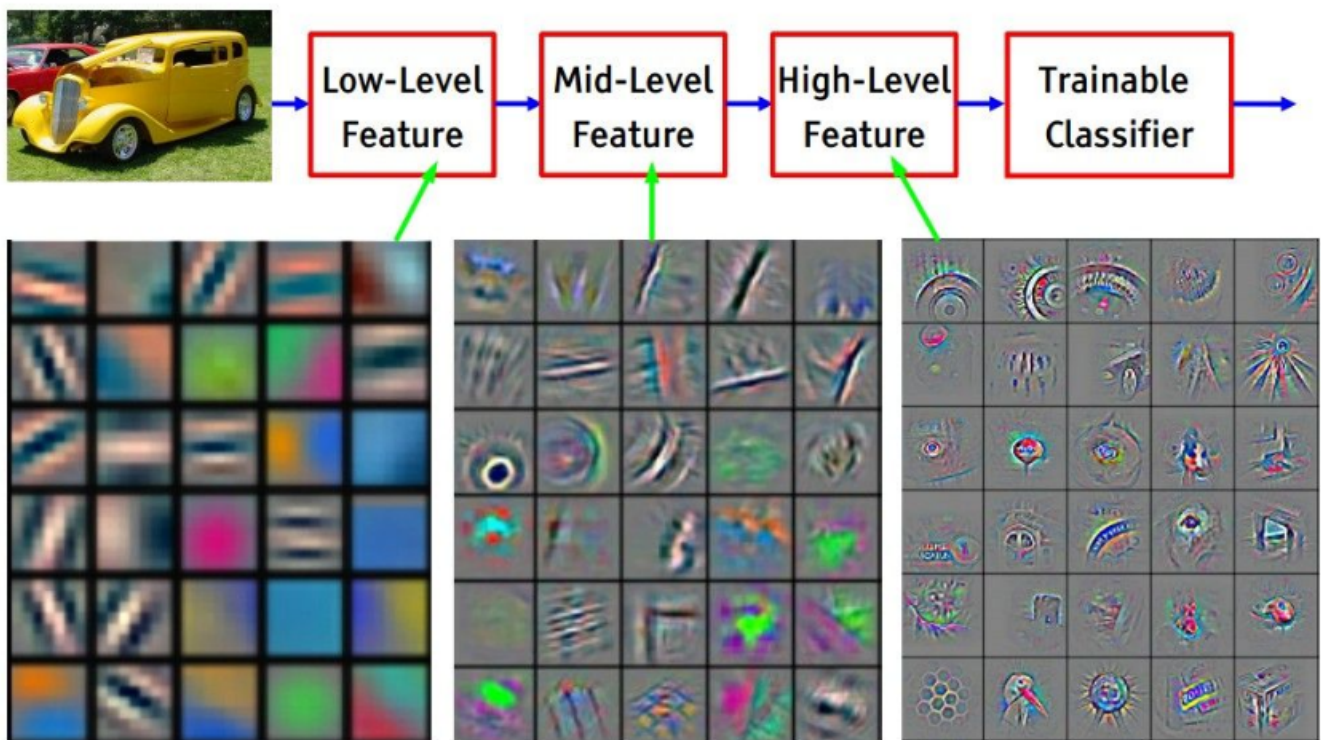


# Image

# Convolved Feature

Convolution operations

The filter's values are tuned through the iterative process of training and after a neural net has trained for certain number of epochs, these filters start to look out for various features in the image. Take the example of face detection using a convolutional neural network. The earlier layers of the network looks for simple features such as edges at different orientations etc. As we progress through the network, the layers start detecting more complex features and when you look at the features detected by the final layers, they almost look like a face.

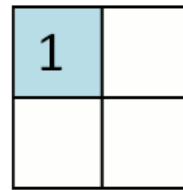
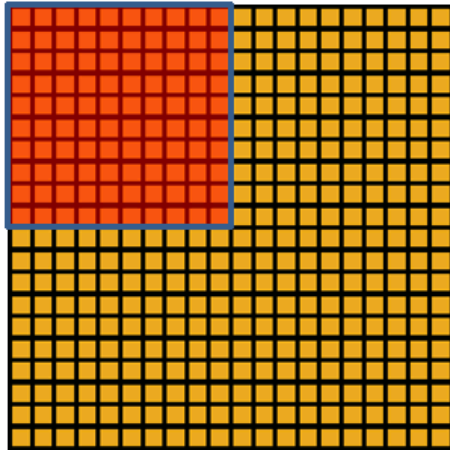


Different features recognised at different layers

Now, let's move on to pooling layers. Pooling layers are used to downsample the image. The image would contain a lot of pixel values and it is typically easy for the network to learn the features if the image size is progressively reduced. Pooling layers help in reducing the number of parameters required and hence, this reduces the computation required. Pooling also helps in avoiding overfitting. There are two types of pooling operation that could be done:

- Max Pooling — Selecting the maximum value
- Average Pooling — Sum all of the values and dividing it by the total number of values

Average pooling is rarely used, you could find max pooling used in most of the examples.



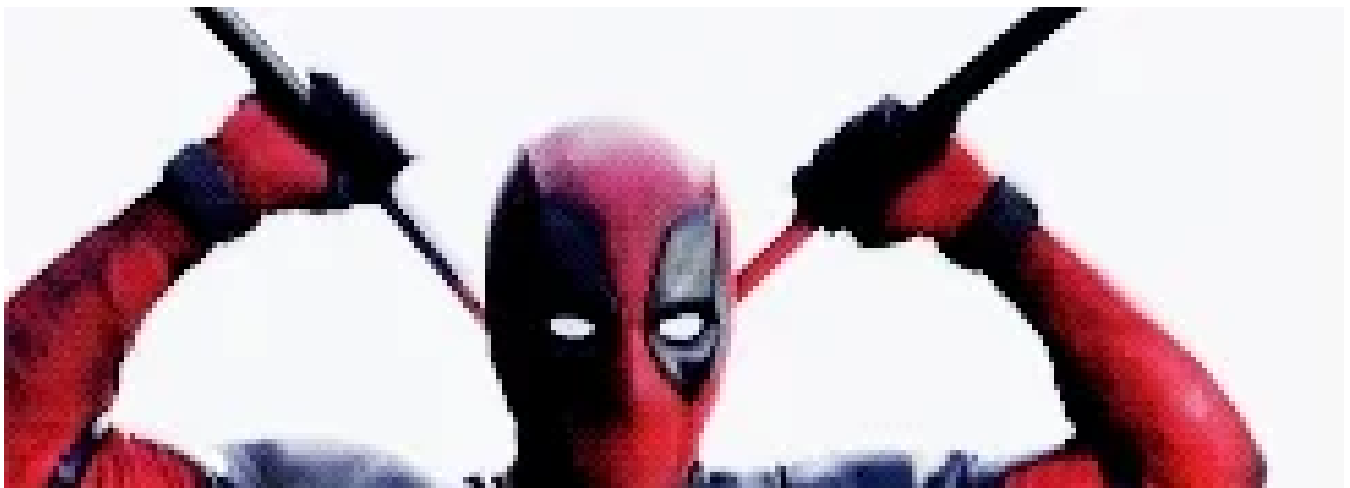
Convolved  
feature

Pooled  
feature

Max Pooling

## Code

Before we start coding, I would like to let you know that the dataset we are going to be using is the MNIST digits dataset and we are going to be using the Keras library with a Tensorflow backend for building the model. Ok, enough. Let's do some coding.





```
1 import keras
2 from keras.datasets import mnist
3 from keras.models import Sequential
4 from keras.layers import Dense, Dropout, Flatten
5 from keras.layers import Conv2D, MaxPooling2D
6 import numpy as np
```

mnist\_imports.py hosted with ♥ by GitHub

[view raw](#)

First, let us do some necessary imports. The keras library helps us build our convolutional neural network. We download the mnist dataset through keras. We import a sequential model which is a pre-built keras model where you can just add the layers. We import the convolution and pooling layers. We also import dense layers as they are used to predict the labels. The dropout layer reduces overfitting and the flatten layer expands a three-dimensional vector into a one-dimensional vector. Finally, we import numpy for matrix operations.

```
1 batch_size = 128
2 num_classes = 10
3 epochs = 12
4
5 # input image dimensions
6 img_rows, img_cols = 28, 28
7
8 # the data, split between train and test sets
9 (x_train, y_train), (x_test, y_test) = mnist.load_data()
10
11 x_train = x_train.reshape(60000, 28, 28, 1)
12 x_test = x_test.reshape(10000, 28, 28, 1)
13
14 print('x_train shape:', x_train.shape)
15 print(x_train.shape[0], 'train samples')
16 print(x_test.shape[0], 'test samples')
17
18 # convert class vectors to binary class matrices
19 y_train = keras.utils.to_categorical(y_train, num_classes)
```

```
20 y_test = keras.utils.to_categorical(y_test, num_classes)
```

mnist\_2.py hosted with ❤ by GitHub

[view raw](#)

Most of the statements in the above code would be trivial, I would just explain some lines of the code. We reshape `x_train` and `x_test` because our CNN accepts only a four-dimensional vector. The value 60000 represents the number of images in the training data, 28 represents the image size and 1 represents the number of channels. The number of channels is set to 1 if the image is in grayscale and if the image is in RGB format, the number of channels is set to 3. We also convert our target values into binary class matrices. To know what binary class matrices look like take a look at the example below.

```
Y = 2 # the value 2 represents that the image has digit 2
```

```
Y = [0,0,1,0,0,0,0,0,0,0] # The 2nd position in the vector is made 1
```

```
# Here, the class value is converted into a binary class matrix
```

```
1 model = Sequential()  
2 model.add(Conv2D(32, kernel_size=(3, 3),  
3                 activation='relu',  
4                 input_shape=(28,28,1)))  
5 model.add(Conv2D(64, (3, 3), activation='relu'))  
6 model.add(MaxPooling2D(pool_size=(2, 2)))  
7 model.add(Dropout(0.25))  
8 model.add(Flatten())  
9 model.add(Dense(128, activation='relu'))  
10 model.add(Dropout(0.5))  
11 model.add(Dense(num_classes, activation='softmax'))
```

mnist\_3.py hosted with ❤ by GitHub

[view raw](#)

We build a sequential model and add convolutional layers and max pooling layers to it. We also add dropout layers in between, dropout randomly switches off some neurons in the network which forces the data to find new paths. Therefore, this reduces overfitting. We add dense layers at the end which are used for class prediction(0–9).

```
1 model.compile(loss=keras.losses.categorical_crossentropy,  
2               optimizer=keras.optimizers.Adadelta(),  
3               metrics=['accuracy'])  
4
```

```

5  model.fit(x_train, y_train,
6          batch_size=batch_size,
7          epochs=epochs,
8          verbose=1,
9          validation_data=(x_test, y_test))
10 score = model.evaluate(x_test, y_test, verbose=0)
11 print('Test loss:', score[0])
12 print('Test accuracy:', score[1])

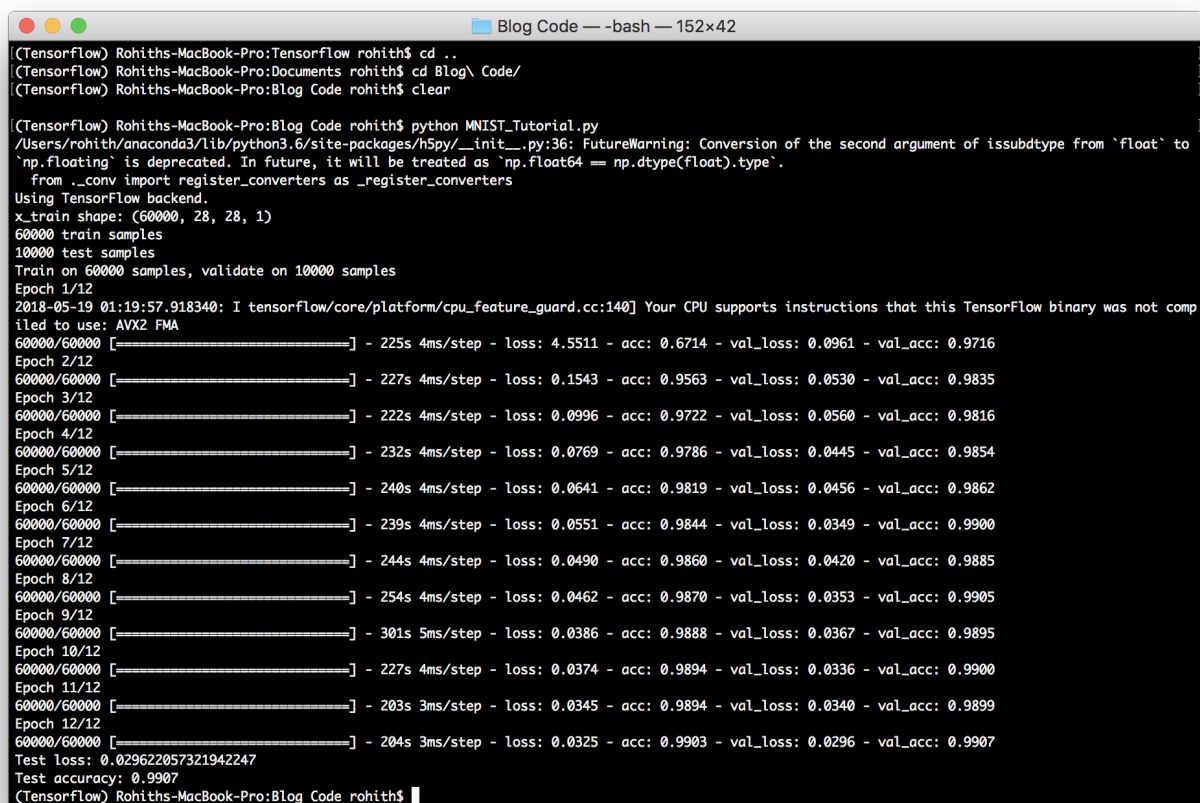
```

mnist\_4.py hosted with ♥ by GitHub

[view raw](#)

We now compile the model with a categorical cross entropy loss function, Adadelata optimizer and an accuracy metric. We then fit the dataset to the model, i.e we train the model for 12 epochs. After training the model, we evaluate the loss and accuracy of the model on the test data and print it.

## Output



```

(TensorFlow) Rohiths-MacBook-Pro:TensorFlow rohith$ cd ..
(TensorFlow) Rohiths-MacBook-Pro:Documents rohith$ cd Blog\ Code\
(TensorFlow) Rohiths-MacBook-Pro:Blog Code rohith$ clear

(TensorFlow) Rohiths-MacBook-Pro:Blog Code rohith$ python MNIST_Tutorial.py
/Users/rohith/anaconda3/lib/python3.6/site-packages/h5py/_init_.py:36: FutureWarning: Conversion of the second argument of issubdtype from `float` to
`np.floating` is deprecated. In future, it will be treated as `np.float64 == np.dtype(float).type`.
  from ..conv import register_converters as _register_converters
Using TensorFlow backend.
x_train shape: (60000, 28, 28, 1)
60000 train samples
10000 test samples
Train on 60000 samples, validate on 10000 samples
Epoch 1/12
2018-05-19 01:19:57.918340: I tensorflow/core/platform/cpu_feature_guard.cc:140] Your CPU supports instructions that this TensorFlow binary was not comp
iled to use: AVX2 FMA
60000/60000 [=====] - 225s 4ms/step - loss: 4.5511 - acc: 0.6714 - val_loss: 0.0961 - val_acc: 0.9716
Epoch 2/12
60000/60000 [=====] - 227s 4ms/step - loss: 0.1543 - acc: 0.9563 - val_loss: 0.0530 - val_acc: 0.9835
Epoch 3/12
60000/60000 [=====] - 222s 4ms/step - loss: 0.0996 - acc: 0.9722 - val_loss: 0.0560 - val_acc: 0.9816
Epoch 4/12
60000/60000 [=====] - 232s 4ms/step - loss: 0.0769 - acc: 0.9786 - val_loss: 0.0445 - val_acc: 0.9854
Epoch 5/12
60000/60000 [=====] - 240s 4ms/step - loss: 0.0641 - acc: 0.9819 - val_loss: 0.0456 - val_acc: 0.9862
Epoch 6/12
60000/60000 [=====] - 239s 4ms/step - loss: 0.0551 - acc: 0.9844 - val_loss: 0.0349 - val_acc: 0.9900
Epoch 7/12
60000/60000 [=====] - 244s 4ms/step - loss: 0.0490 - acc: 0.9860 - val_loss: 0.0420 - val_acc: 0.9885
Epoch 8/12
60000/60000 [=====] - 254s 4ms/step - loss: 0.0462 - acc: 0.9870 - val_loss: 0.0353 - val_acc: 0.9905
Epoch 9/12
60000/60000 [=====] - 301s 5ms/step - loss: 0.0386 - acc: 0.9888 - val_loss: 0.0367 - val_acc: 0.9895
Epoch 10/12
60000/60000 [=====] - 227s 4ms/step - loss: 0.0374 - acc: 0.9894 - val_loss: 0.0336 - val_acc: 0.9900
Epoch 11/12
60000/60000 [=====] - 203s 3ms/step - loss: 0.0345 - acc: 0.9894 - val_loss: 0.0340 - val_acc: 0.9899
Epoch 12/12
60000/60000 [=====] - 204s 3ms/step - loss: 0.0325 - acc: 0.9903 - val_loss: 0.0296 - val_acc: 0.9907
Test loss: 0.029622057321942247
Test accuracy: 0.9907
(TensorFlow) Rohiths-MacBook-Pro:Blog Code rohith$

```

## Conclusion

Convolutional neural networks do have some shortcomings pointed out by Geoffrey Hinton. He posited that his capsule networks are the way to go if we are looking to achieve human-level accuracy in the domain of computer vision. But, as of now, CNNs

seem to be doing really well. Please let me know if you found this article to be useful, thank you :)

[Machine Learning](#)[Convolutional Network](#)[Deep Learning](#)[Keras](#)[About](#)[Help](#)[Legal](#)