# Convolutional Neural Networks: An Intro Tutorial

Derrick Mwiti
May 8, 2018 · 11 min read

A Convolutional Neural Network (CNN) is a multilayered neural network with a special architecture to detect complex features in data. CNNs have been used in image recognition, powering vision in robots, and for self-driving vehicles.

In this article, we're going to build a CNN capable of classifying images. An image classifier CNN can be used in myriad ways, to classify cats and dogs, for example, or to detect if pictures of the brain contain a tumor. This post will be at an introductorylevel, and no domain expertise is required. However, we assume that the reader has a basic understanding of Artificial Neural Networks (ANN).

Once a CNN is built, it can be used to classify the contents of different images. All we have to do is feed those images into the model. Just like ANNs, CNNs are inspired by the workings of the human brain. CNNs are able to classify images by detecting features, similar to how the human brain detects features to identify objects.

Before we dive in and build the model, let's understand some concepts of CNNs and the steps of building one.

## How do CNNs work?

Images are made up of pixels. Each pixel is represented by a number between 0 and 255. Therefore each image has a digital representation which is how computers are able to work with images.

### 1. Convolution

A convolution is a combined integration of two functions that shows you how one function modifies the other.

$$\begin{aligned}(f * g)(t) &\overset{\text{def}}{=} \int_{-\infty}^{\infty} f(\tau)g(t-\tau)\,d\tau \\ &= \int_{-\infty}^{\infty} f(t-\tau)g(\tau)\,d\tau.\end{aligned}$$

*[The convolution function. Source: Wikipedia]*

There are three important items to mention in this process: the input image, the feature detector, and the feature map. The input image is the image being detected. The feature detector is a matrix, usually 3x3 (it could also be 7x7). A **feature detector** is also referred to as a kernel or a filter.

Intuitively, the matrix representation of the input image is multiplied element-wise with the feature detector to produce a feature map, also known as a convolved feature or an activation map. The aim of this step is to reduce the size of the image and make processing faster and easier. Some of the features of the image are lost in this step.

However, the main features of the image that are important in image detection are retained. These features are the ones that are unique to identifying that specific object.

For example each animal has unique features that enable us to identify it. The way we prevent loss of image information is by having many feature maps. Each feature map detects the location of certain features in the image.

### 2. Apply the ReLu (Rectified Linear Unit)

In this step we apply the rectifier function to increase non-linearity in the CNN. Images are made of different objects that are not linear to each other. Without applying this function the image classification will be treated as a linear problem while it is actually a non-linear one.

### 3. Pooling

Spatial invariance is a concept where the location of an object in an image doesn't affect the ability of the neural network to detect its specific features. Pooling enables the CNN to detect features in various images irrespective of the difference in lighting in the pictures and different angles of the images.

There are different types of pooling, for example, max pooling and min pooling. Max pooling works by placing a matrix of 2x2 on the feature map and picking the largest value in that box. The 2x2 matrix is moved from left to right through the entire feature map picking the largest value in each pass.

These values then form a new matrix called a pooled feature map. Max pooling works to preserve the main features while also reducing the size of the image. This helps reduce overfitting, which would occur if the CNN is given too much information, especially if that information is not relevant in classifying the image.

### 4. Flattening

Once the pooled featured map is obtained, the next step is to flatten it. Flattening involves transforming the entire pooled feature map matrix into a single column which is then fed to the neural network for processing.

### 5. Full connection

After flattening, the flattened feature map is passed through a neural network. This step is made up of the input layer, the fully connected layer, and the output layer. The fully connected layer is similar to the hidden layer in ANNs but in this case it's fully connected. The output layer is where we get the predicted classes. The information is passed through the network and the error of prediction is calculated. The error is then backpropagated through the system to improve the prediction.

The final figures produced by the neural network don't usually add up to one. However, it is important that these figures are brought down to numbers between zero and one, which represent the probability of each class. This is the role of the Softmax function.

$$\sigma : \mathbb{R}^K \to (0,1)^K$$
$$\sigma(\mathbf{z})_j = \frac{e^{z_j}}{\sum_{k=1}^K e^{z_k}} \quad \text{for } j = 1, \ldots, K.$$

*[The Softmax function. Source: Wikipedia]*

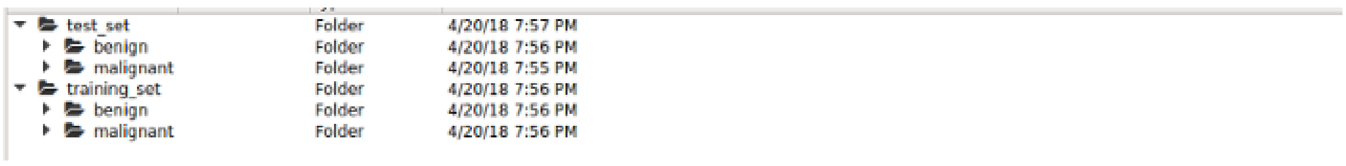
. . .

Machine learning models are moving closer and closer to edge devices. Fritz AI is here to help with this transition. Explore our suite of developer tools that makes it easy to teach devices to see, hear, sense, and think.

. . .

## Implementation with Keras using TensorFlow backend

Now let's write the code that can classify images. For this exercise it's advisable to arrange the the folders that contain images as shown below. We separate images into folders and give them their appropriate names, i.e the training set and the test set. This makes it easier to import the images into Keras. Make sure that the working directory has permissions to access the images.



```
▾ test_set        Folder   4/20/18 7:57 PM
   ▸ benign       Folder   4/20/18 7:56 PM
   ▸ malignant    Folder   4/20/18 7:55 PM
▾ training_set    Folder   4/20/18 7:56 PM
   ▸ benign       Folder   4/20/18 7:56 PM
   ▸ malignant    Folder   4/20/18 7:56 PM
```

## 1) Setup

In this step we need to import Keras and other packages that we're going to use in building the CNN. Import the following packages:

- *Sequential* is used to initialize the neural network.
- *Convolution2D* is used to make the convolutional network that deals with the images.
- *MaxPooling2D* layer is used to add the pooling layers.
- *Flatten* is the function that converts the pooled feature map to a single column that is passed to the fully connected layer.
- *Dense* adds the fully connected layer to the neural network.

```
from keras.models import Sequential

from keras.layers import Convolution2D

from keras.layers import MaxPooling2D

from keras.layers import Flatten

from keras.layers import Dense
```

## 2) Initializing the neural network

To initialize the neural network we create an object of the Sequential class.

```
classifier = Sequential()
```

## 3) Convolution

To add the convolution layer, we call the `add` function with the classifier object and pass in `Convolution2D` with parameters. The first argument `nb_filter. nbfilter` is the number of feature detectors that we want to create. The second and third parameters are dimensions of the feature detector matrix.

It's common practice to start with 32 feature detectors for CNNs. The next parameter is `input_shape` which is the shape of the input image. The images will be converted into this shape during preprocessing. If the image is black and white it will be converted into a 2D array and if the image is colored it will be converted into a 3D array.

In this case, we'll assume that we are working with colored images. `Input_shape` is passed in a tuple with the number of channels, which is 3 for a colored image, and the dimensions of the 2D array in each channel. If you are not using a GPU it's advisable to use lower dimensions to reduce the computation time. When using a CPU, 64 by 64 dimensions performs well. The final parameter is the activation function. Classifying images is a nonlinear problem. So we use the rectifier function to ensure that we don't have negative pixel values during computation. That's how we achieve non-linearity.

```
classifier.add(Convolution2D(32, 3, 3, input_shape = (256, 256, 3),
activation='relu'))
```

## 4) Pooling

In this step we reduce the size of the feature map. Generally we create a pool size of 2x2 for max pooling. This enables us to reduce the size of the feature map while not losing important image information.

`classifier.add(MaxPooling2D(pool_size=(2,2)))`

## 5) Flattening

In this step, all the pooled feature maps are taken and put into a single vector. The `Flatten` function flattens all the feature maps into a single column.

`classifier.add(Flatten())`

## 6) Full connection

The next step is to use the vector we obtained above as the input for the neural network by using the `Dense` function in Keras. The first parameter is `output_dim` which is the number of nodes in the hidden layer. You can determine the most appropriate number through experimentation. The higher the number of dimensions the more computing resources you will need to fit the model. A common practice is to pick the number of nodes in powers of two. The second parameter is the activation function. We usually use the `ReLu` activation function in the hidden layer.

`classifier.add(Dense(output_dim = 128, activation='relu'))`

The next layer we have to add is the output layer. In this case, we'll use the `sigmoid` activation function since we expect a binary outcome. If we expected more than two outcomes we would use the `softmax` function.

The `output_dim` here is 1 since we just expect the predicted probabilities of the classes.

```
classifier.add(Dense(output_dim=1, activation='sigmoid'))
```

. . .

Machine learning is rapidly moving closer to where data is collected — edge devices. Subscribe to the Fritz AI Newsletter to learn more about this transition and how it can help scale your business.

. . .

## 7) Compiling CNN

We then compile the CNN using the `compile` function. This function expects three parameters: the optimizer, the loss function, and the metrics of performance.The optimizer is the gradient descent algorithm we are going to use. We use the `binary_crossentropy` loss function since we are doing a binary classification.

```
classifier.compile(optimizer='adam', loss='binary_crossentropy',metrics=
['accuracy'])
```

## 8) Fitting the CNN

We are going to preprocess the images using Keras to prevent overfitting. This processing is known as image augmentation. The Keras utility we use for this purpose is `ImageDataGenerator`.

```
from keras.preprocessing.image import ImageDataGenerator
```

This function works by flipping, rescaling, zooming, and shearing the images. The first argument `rescale` ensures the images are rescaled to have pixel values between zero and one. `horizontal_flip=True` means that the images will be flipped horizontally. All these actions are part of the image augmentation.

```
train_datagen = ImageDataGenerator(rescale=1./255, shear_range=0.2,
zoom_range=0.2, horizontal_flip=True)
```

We then use the `ImageDataGenerator` function to rescale the pixels of the test set so that they are between zero and one. Since this is the test data and not the training data we don't have to take image augmentation steps.

```
test_datagen = ImageDataGenerator(rescale=1./255)
```

The next thing we need to do is create the training set. We do this by using `train_datagen` that we just created above and the `flow_from_directory` function. The `flow_from_directory` function enables us to retrieve the images of our training set from the current working directory. The first parameter is the path to the training set.

The second parameter is the `target_size`, which is the size of the image that the CNN should expect. We have already specified this above as `256x256`, so we shall use the same for this parameter. The `batch_size` is the number of images that will go through the network before the weights are updated. The `class_mode` parameter indicates whether the classification is binary or not.

```
training_set = train_datagen.flow_from_directory('training_set', target_size=(256,
256), batch_size=32, class_mode='binary')
```

Now we will create the test set with similar parameters as above.

```
test_set = test_datagen.flow_from_directory('test_set', target_size=(64, 64),
batch_size=32, class_mode='binary')
```

Finally, we need to fit the model to the training dataset and test its performance with the test set. We achieve this by calling the `fit_generator` function on the classifier object. The first argument it takes is the training set. The second argument is the number of arguments in our training set. `Epochs` is the number of epochs we want to use to train the CNN. `Validation_data` is the test data set. `nb_val_samples` is the number of images in the test set.

```
classifier.fit_generator(training_set, steps_per_epoch=5000, epochs=25,
validation_data=test_set, nb_val_samples=1000)
```

## 9) Making a single prediction

Now that the model is fitted, we can use the `predict` method to make predictions using new images. In order to do this we need to preprocess our images before we pass them to the predict method. To achieve this we'll use some functions from numpy. We also need to import the `image` module from Keras to allow us to load in the new images.

```
import numpy as np
from keras.preprocessing import image
```

The next step is to load the image that we would like to predict. To accomplish this we use the `load_img` function from the `image` module. The first argument this function takes is the path to the location of the image and the second argument is the size of the image. The size of the image should be the same as the size used during the training process.

```
test_image = image.load_img('brain_image1.jpg', target_size=(256, 256))
```

As noted earlier, we're using 3 channels because our images are color and therefore need to transform this image into a 3D array. To do this we use the `img_to_array` function from the `image`module.

```
test_image = image.img_to_array(test_image)
```

We now have an image with three dimensions. However we are not yet ready to make the predictions because the `predict` method expects four dimensions. The fourth dimension corresponds to the batch size. This is because in neural networks the data to be predicted is usually passed in as a batch. In this case we have one batch of one input image. We use the `expand_dims` method from numpy to add this new dimension. It takes the first parameter as the test image we are expanding, and the second parameter is the position of the dimension we are adding. The `predict` method expects this new dimension in the first position, which corresponds to axis 0.

```
test_image = np.expand_dims(test_image, axis=0)
```

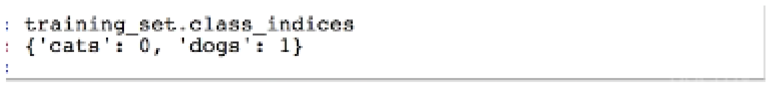
Now we use the `predict` method to predict which class the image belongs to.

```
prediction = classifier.predict(test_image)
```

After running this function we'll get the result: either one or zero. However we don't know which value represents which class. To find out, we use the `class_indices` attribute of the training set.

```
training_set.class_indices
```

Assuming you had done this classification for cats and dogs you would get the following output

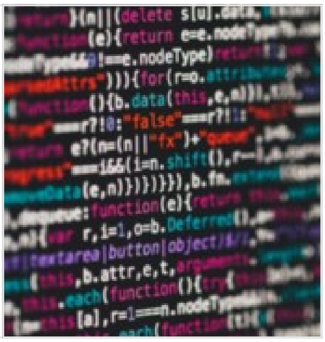


```
: training_set.class_indices
: {'cats': 0, 'dogs': 1}
:
```

## Conclusion

We have now written a deep learning model that can be used to classify images into different classes. You can use this model to do any image classification. All you have to do is put the training mages and the test images in their correct folders as shown in this article. Once that is done you will be ready to do your image classification.

Congratulations for staying with me this far. If you would like to learn more about CNNs you can check out the Keras documentation.



**Discuss this post on Hacker News.**

. . .

*Editor's Note:* ***Heartbeat*** *is a contributor-driven online publication and community dedicated to exploring the emerging intersection of mobile app development and machine learning. We're committed to supporting and inspiring developers and engineers from all walks of life.*

*Editorially independent, Heartbeat is sponsored and published by* ***Fritz AI****, the machine learning platform that helps developers teach devices to see, hear, sense, and think. We pay our contributors, and we don't sell ads.*

*If you'd like to contribute, head on over to our* ***call for contributors****. You can also sign up to receive our weekly newsletters (****Deep Learning Weekly*** *and the* ***Fritz AI Newsletter****), join us on* ***Slack****, and follow Fritz AI on* ***Twitter*** *for all the latest in mobile machine learning.*

Machine Learning Keras Neural Networks Heartbeat Guides And Tutorials