

This is your **last** free member-only story this month. [Upgrade for unlimited access.](#)

Akash Chauhan

Follow

Jul 10, 2020 · 8 min read ·

Convolutional Neural Networks for Multiclass Image Classification — A Beginners Guide to Understand CNN

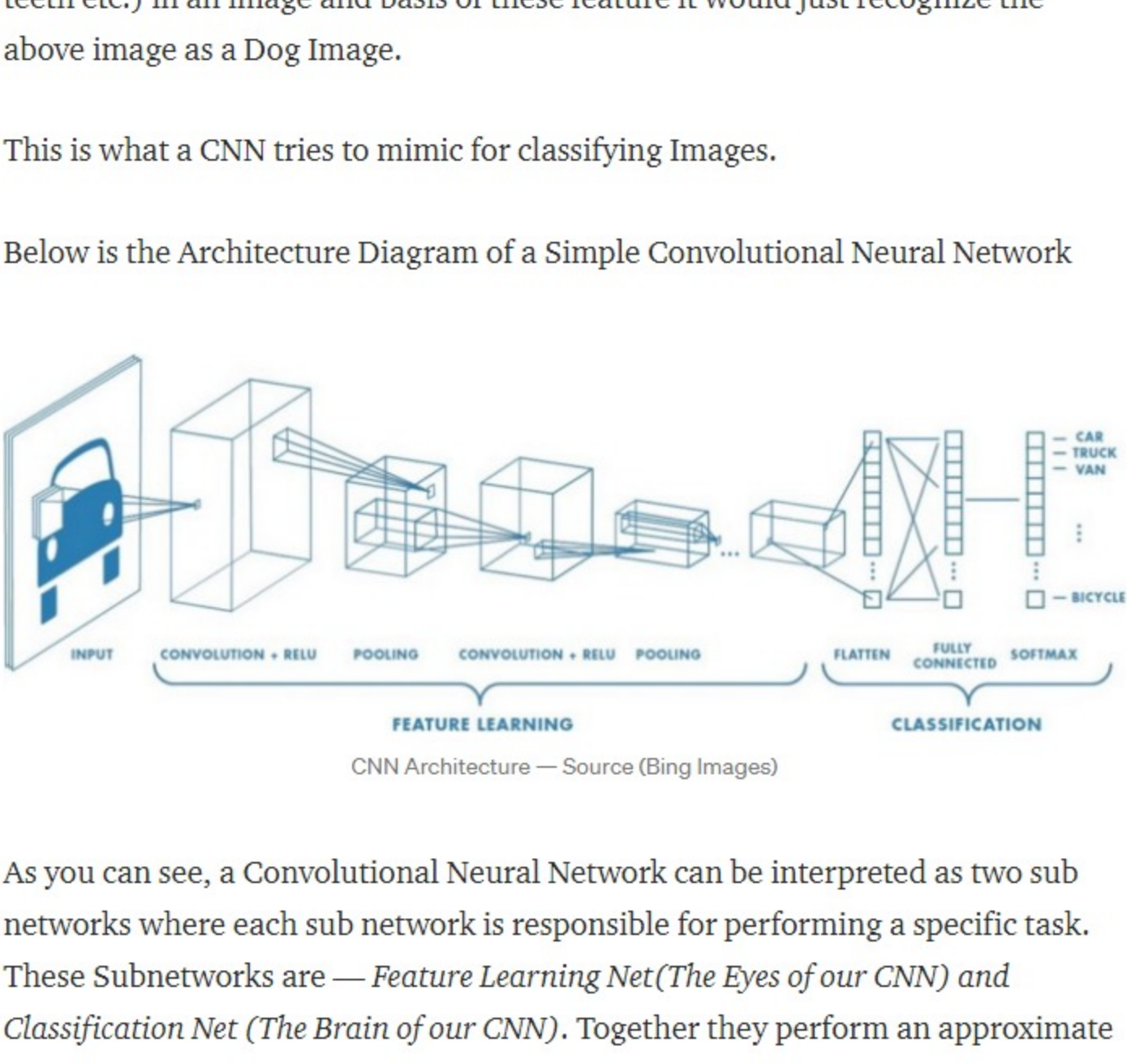
Convolutional Neural Network (**ConvNet** or **CNN**) is a class of deep neural networks most commonly used for analyzing visual imagery. Convolution layers are the building blocks of the CNNs.

A convolution is the simple application of a filter to an input that results in an activation. Repeated application of the same filter to an input results in a map of activations called a feature map, indicating the locations and strength of a detected feature in an input, such as an image.

What makes CNNs so powerful and useful is that they can generate excellent predictions with minimal image preprocessing. Also, the CNNs are immune to spatial variance and hence are able to detect features anywhere in the input images. This article will let the readers understand how CNNs work along with its Python implementation using Tensorflow and Keras libraries to solve a multiclass classification problem.

CNN Architecture

Before talking about CNN Architecture, lets understand how human brain actually recognizes an image using an example. Try and recognize this image..



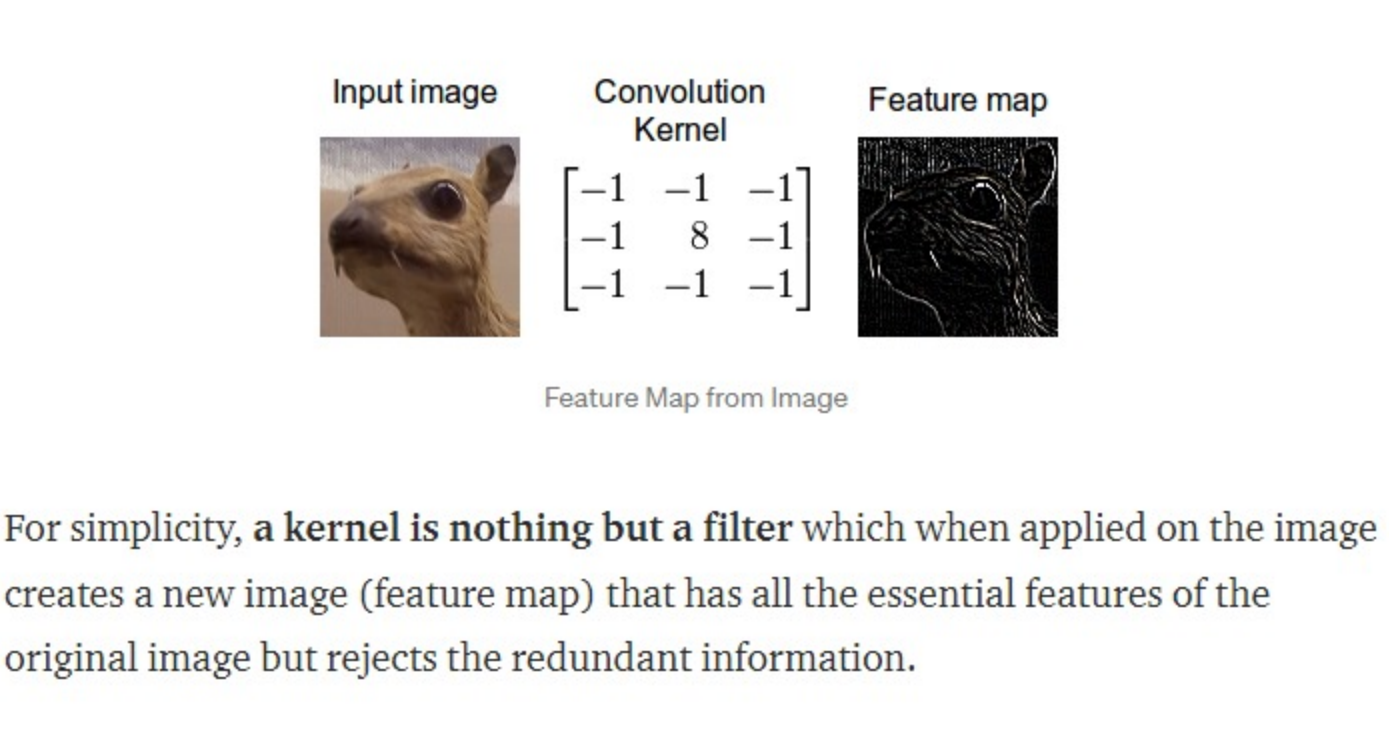
Example Image

We all know its an image of dog and even with just a glimpse of this image or any other similar image we would always know that this a dog, but how do we know this?? How are we so sure and correct all the time?

The reason is that with every evolution step in Humankind, our brain has learned to identify certain key features (big ears, hairy face, long mouth, large teeth etc.) in an image and basis of these feature it would just recognize the above image as a Dog Image.

This is what a CNN tries to mimic for classifying Images.

Below is the Architecture Diagram of a Simple Convolutional Neural Network



CNN Architecture — Source (Bing Images)

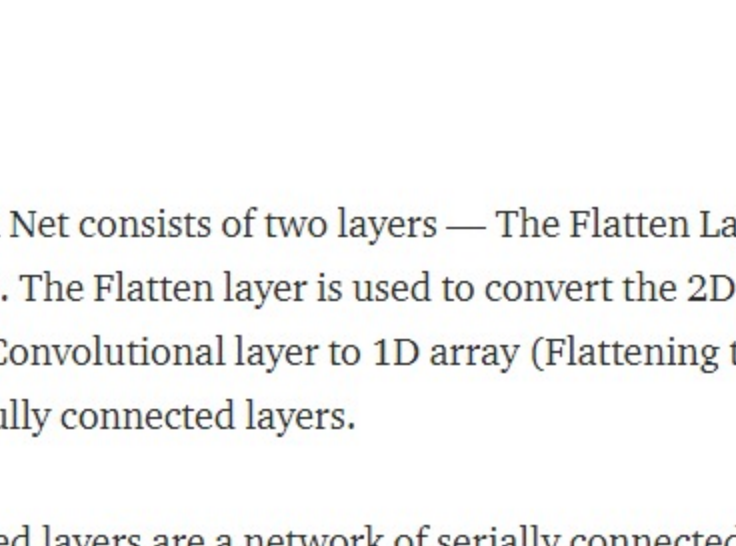
As you can see, a Convolutional Neural Network can be interpreted as two sub networks where each sub network is responsible for performing a specific task. These Subnetworks are — *Feature Learning Net (The Eyes of our CNN)* and *Classification Net (The Brain of our CNN)*. Together they perform an approximate function of how a Human Brain classifies the images.

- Feature Learning** — The Feature Learning part mainly consist of Convolutional Layers and Pooling Layers . The number of Convolutional and Pooling Layers in a CNN are usually more than one and are directly dependent on the nature of the classification problem (More complex problem would require more Convolutional and Pooling layers for feature extractions).
- Classification** — The classification part is responsible for classifying the images to their respective categories based on the features (Feature Maps) that Feature Learning part has extracted (created) from the image. The Classification part usually consist of a Flatten Layer and a network of Fully Connected Hidden Layers.

Feature Learning

The Feature Learning Net consists of two layers — The Convolution Layer and The Pooling Layer. Lets talk about the Convolution Layer first.

The **Convolution Layer** or sometimes referred to as a decomposition layer is responsible for decomposing the Image into various feature maps using different kernels. The resulting Feature Map is a convolved Image generated from our original Image. Lets understand this with the below example-

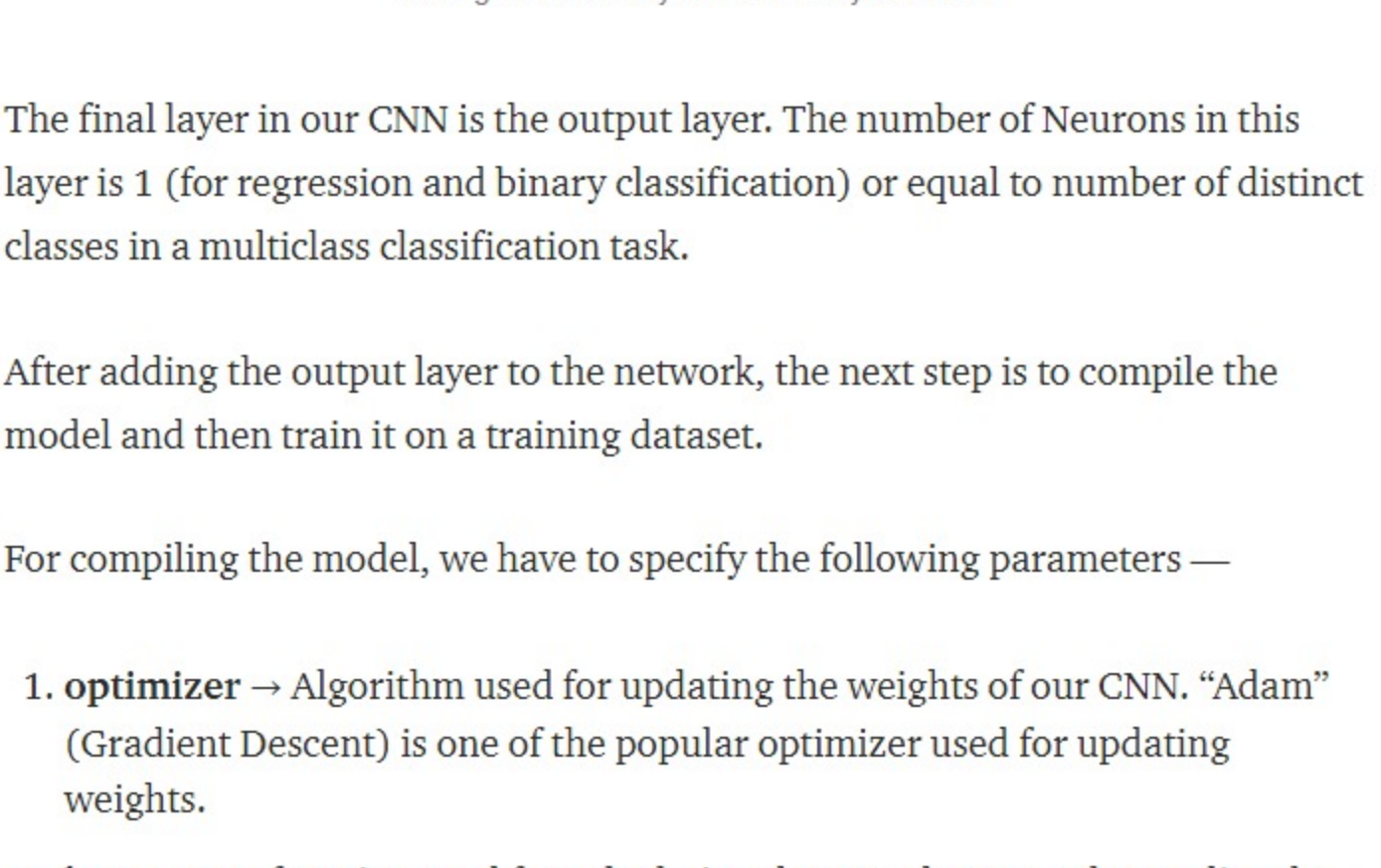


Feature Map from Image

For simplicity, a **kernel** is **nothing but** a filter which when applied on the image creates a new image (feature map) that has all the essential features of the original image but rejects the redundant information.

This is why **Convolution** is also referred to as **Decomposition** sometime.

The next layer in our network is the **Pooling Layer** which is responsible for approximating the feature maps that were created using the Convolutional layers. The Pooling Layer is also responsible for accounting any spatial variance that might effect the performance of our network in Image Recognition.



Max Pooling

This is how **Pooling Layer** functions in a **ConvNet**. There are many types of Pooling layers (Mean, Max etc.). For our **ConvNet** we are using **Max Pooling**.

The following code will help you understand how to add a convolution layer and a Pooling Layer to our CNN.

```
1 import tensorflow as tf
2 from keras.preprocessing.image import ImageDataGenerator
3
4 # creating a sequential model
5 cnn = tf.keras.models.Sequential()
6
7 # adding convolution layer to network
8 cnn.add(tf.keras.layers.Conv2D(filters=32, kernel_size=3, activation='relu', input_shape=[64, 64,
9
10 # adding pooling layer to network
11 cnn.add(tf.keras.layers.MaxPool2D(pool_size=2, strides=2))
```

Adding Convolutional & Pooling Layer to CNN

Following are the arguments of the Conv2D function-

- filters** — Number of different filters (feature detectors) that will be applied on the original image to create feature maps. Different types of filters are Edge Detection Filter, Blur Filter etc.
- kernel_size** — Dimension of the convolution filter (n x n) matrix.
- Activation** — The activation function for the neurons. A general thumb of rule is to use a **Rectifier Linear Unit (Relu)** function as an activation function for every layer besides the output layer. The Relu function also adds non linearity to our network which is highly required to eliminate any linear relationships that does exist in the feature maps.
- Input Layer** — Takes the shape of the Input Images and number of channels (3 for color and 1 for B/W image).

Following are the arguments of the MaxPool2D function-

- pool_size** — Dimension of pooling matrix (m x m)
- strides** — The number of rows and columns traversed per slide.

Classification

The Classification Net consists of two layers — The Flatten Layer and The Fully Connected Layers. The Flatten layer is used to convert the 2D output array from Pooling Layer or Convolutional layer to 1D array (Flattening the input) before feeding it to the fully connected layers.

The fully connected layers are a network of serially connected dense layers that would be used for classification. In a fully connected network every neuron from layer1 is connected to every neuron in layer2. Usually for Computer Vision Application, the dense layers have large numbers of neurons (256, 128 etc.) for achieving higher accuracy in our predictions.

The following code can be used to add a Flatten and a network of fully connected layers to our CNN.

```
1 # adding a flatten layer to CNN
2 cnn.add(tf.keras.layers.Flatten())
3
4 # adding fully connected layers
5 cnn.add(tf.keras.layers.Dense(128, activation='relu'))
6 cnn.add(tf.keras.layers.Dense(64, activation='relu'))
7
8 # output layer -> 6 Neurons for 6 different classes
9 # activation function used for multiclass classification is softmax, for binary use sigmoid as act
10 cnn.add(tf.keras.layers.Dense(6, activation='softmax'))
```

Adding Flatten & Fully Connected Layer to CNN

The final layer in our CNN is the output layer. The number of Neurons in this layer is 1 (for regression and binary classification) or equal to number of distinct classes in a multiclass classification task.

After adding the output layer to the network, the next step is to compile the model and then train it on a training dataset.

For compiling the model, we have to specify the following parameters —

- optimizer** → Algorithm used for updating the weights of our CNN. “Adam” (Gradient Descent) is one of the popular optimizer used for updating weights.
- loss** → Cost function used for calculating the error between the predicted & actual value. In our case we will be using “categorical_crossentropy” since we are dealing with multiclass classification. In case of binary classification we have to use “binary_crossentropy” as loss function.
- metrics** → Evaluation metric for checking performance of our model.

For fitting the model, we have to specify the following parameters —

- batch_size** → Number of images that will be used by to train our CNN model before updating the weights using back propagation.
- epochs** → An **epoch** is a measure of the number of times all of the training images are used once to update the weights.

Till now we have learnt about the different components of a Convolutional Neural Network. Let us now develop our own CNN for image classification.

The problem at our hand is image data of Nature Scenes around the world. The Data contains around 25k images of size 150x150 distributed under 6 categories.

{'buildings' -> 0, 'forest' -> 1, 'glacier' -> 2, 'mountain' -> 3, 'sea' -> 4, 'street' -> 5 }

Our goal is to develop a CNN that can accurately classify the new images into one of these category.

Developing a CNN for Multiclass Image Classification

Before training our CNN on training set images, we first have to apply image augmentation. The reason for performing image augmentation is to eliminate any chance of overfitting our model (training accuracy >> testing accuracy).

Image Augmentation can be applied using **ImageDataGenerator** function of **keras.preprocessing.image** class. Along with augmentation, we also need to rescale our images (every pixel value becomes a value between 0 & 1).

The following code can be used to develop a CNN for image classification.

```
1 # import statements
2 import tensorflow as tf
3 from keras.preprocessing.image import ImageDataGenerator
4 import numpy as np
5
6 # loading training data
7 train_datagen = ImageDataGenerator(
8     rescale=1./255,
9     shear_range=0.2,
10    zoom_range=0.2,
11    horizontal_flip=True)
12 train_generator = train_datagen.flow_from_directory(
13     '/kaggle/input/intel-image-classification/seg_train/seg_train',
14     target_size=(64, 64),
15     batch_size=32,
16     class_mode='categorical')
17
18 # loading testing data
19 test_datagen = ImageDataGenerator(rescale=1./255)
20 test_generator = test_datagen.flow_from_directory(
21     '/kaggle/input/intel-image-classification/seg_test/seg_test',
22     target_size=(64, 64),
23     batch_size=32,
24     class_mode='categorical')
25
26 # initialising sequential model and adding layers to it
27 cnn = tf.keras.models.Sequential()
28 cnn.add(tf.keras.layers.Conv2D(filters=48, kernel_size=3, activation='relu', input_shape=[64, 64,
29     cnn.add(tf.keras.layers.MaxPool2D(pool_size=2, strides=2))
30     cnn.add(tf.keras.layers.Conv2D(filters=48, kernel_size=3, activation='relu'))
31     cnn.add(tf.keras.layers.MaxPool2D(pool_size=2, strides=2))
32     cnn.add(tf.keras.layers.Conv2D(filters=32, kernel_size=3, activation='relu'))
33     cnn.add(tf.keras.layers.MaxPool2D(pool_size=2, strides=2))
34     cnn.add(tf.keras.layers.Flatten())
35     cnn.add(tf.keras.layers.Dense(128, activation='relu'))
36     cnn.add(tf.keras.layers.Dense(64, activation='relu'))
37     cnn.add(tf.keras.layers.Dense(6, activation='softmax'))
38
39 # finally compile and train the cnn
40 cnn.compile(optimizer="adam", loss="categorical_crossentropy", metrics=["accuracy"])
41 cnn.fit(x=train_generator, validation_data=test_generator, epochs=30)
```

Code for developing a CNN

The following is the output of the model while training.

```
Epoch 1/30
439/439 [=====] - 52s 119ms/step - loss: 1.1012 - accuracy: 0.5655 - val_loss: 0.9319 - val_accuracy: 0.6333
Epoch 2/30
439/439 [=====] - 53s 121ms/step - loss: 0.8520 - accuracy: 0.6773 - val_loss: 0.7698 - val_accuracy: 0.7140
Epoch 3/30
439/439 [=====] - 52s 119ms/step - loss: 0.7336 - accuracy: 0.7292 - val_loss: 0.7235 - val_accuracy: 0.7243
Epoch 4/30
439/439 [=====] - 53s 120ms/step - loss: 0.6590 - accuracy: 0.7579 - val_loss: 0.6601 - val_accuracy: 0.7593
Epoch 5/30
439/439 [=====] - 52s 119ms/step - loss: 0.6014 - accuracy: 0.7832 - val_loss: 0.7235 - val_accuracy: 0.7777
Epoch 6/30
439/439 [=====] - 52s 118ms/step - loss: 0.5648 - accuracy: 0.7964 - val_loss: 0.6243 - val_accuracy: 0.7740
Epoch 7/30
439/439 [=====] - 52s 119ms/step - loss: 0.5266 - accuracy: 0.8109 - val_loss: 0.5804 - val_accuracy: 0.7880
Epoch 8/30
439/439 [=====] - 53s 120ms/step - loss: 0.4994 - accuracy: 0.8195 - val_loss: 0.5232 - val_accuracy: 0.8097
Epoch 9/30
439/439 [=====] - 52s 120ms/step - loss: 0.4811 - accuracy: 0.8253 - val_loss: 0.5347 - val_accuracy: 0.8140
Epoch 10/30
439/439 [=====] - 53s 120ms/step - loss: 0.4729 - accuracy: 0.8292 - val_loss: 0.5352 - val_accuracy: 0.8113
Epoch 11/30
439/439 [=====] - 52s 119ms/step - loss: 0.4439 - accuracy: 0.8407 - val_loss: 0.5560 - val_accuracy: 0.8010
Epoch 12/30
439/439 [=====] - 52s 119ms/step - loss: 0.4406 - accuracy: 0.8409 - val_loss: 0.5026 - val_accuracy: 0.8273
Epoch 13/30
439/439 [=====] - 54s 124ms/step - loss: 0.4236 - accuracy: 0.8459 - val_loss: 0.5692 - val_accuracy: 0.8107
Epoch 14/30
439/439 [=====] - 52s 118ms/step - loss: 0.4188 - accuracy: 0.8499 - val_loss: 0.4980 - val_accuracy: 0.8190
Epoch 15/30
439/439 [=====] - 52s 119ms/step - loss: 0.3954 - accuracy: 0.8590 - val_loss: 0.5697 - val_accuracy: 0.7963
Epoch 16/30
439/439 [=====] - 54s 123ms/step - loss: 0.3771 - accuracy: 0.8643 - val_loss: 0.5335 - val_accuracy: 0.8127
Epoch 17/30
439/439 [=====] - 53s 120ms/step - loss: 0.3766 - accuracy: 0.8616 - val_loss: 0.5396 - val_accuracy: 0.8120
Epoch 18/30
439/439 [=====] - 52s 118ms/step - loss: 0.3639 - accuracy: 0.8666 - val_loss: 0.5345 - val_accuracy: 0.8163
Epoch 19/30
439/439 [=====] - 52s 119ms/step - loss: 0.3561 - accuracy: 0.8705 - val_loss: 0.4960 - val_accuracy: 0.8313
Epoch 20/30
439/439 [=====] - 52s 119ms/step - loss: 0.3425 - accuracy: 0.8764 - val_loss: 0.5402 - val_accuracy: 0.8240
Epoch 21/30
439/439 [=====] - 52s 118ms/step - loss: 0.3405 - accuracy: 0.8746 - val_loss: 0.5499 - val_accuracy: 0.8310
Epoch 22/30
439/439 [=====] - 52s 119ms/step - loss: 0.3257 - accuracy: 0.8825 - val_loss: 0.5232 - val_accuracy: 0.8253
Epoch 23/30
439/439 [=====] - 51s 117ms/step - loss: 0.3154 - accuracy: 0.8835 - val_loss: 0.5071 - val_accuracy: 0.8340
Epoch 24/30
439/439 [=====] - 51s 117ms/step - loss: 0.3121 - accuracy: 0.8858 - val_loss: 0.5404 - val_accuracy: 0.8143
Epoch 25/30
439/439 [=====] - 52s 118ms/step - loss: 0.3082 - accuracy: 0.8847 - val_loss: 0.5662 - val_accuracy: 0.8217
Epoch 26/30
439/439 [=====] - 52s 119ms/step - loss: 0.2939 - accuracy: 0.8950 - val_loss: 0.5330 - val_accuracy: 0.8230
Epoch 27/30
439/439 [=====] - 52s 117ms/step - loss: 0.2821 - accuracy: 0.8970 - val_loss: 0.5394 - val_accuracy: 0.8253
Epoch 28/30
439/439 [=====] - 52s 118ms/step - loss: 0.2794 - accuracy: 0.8966 - val_loss: 0.5438 - val_accuracy: 0.8360
Epoch 29/30
439/439 [=====] - 51s 117ms/step - loss: 0.2709 - accuracy: 0.9017 - val_loss: 0.5954 - val_accuracy: 0.8170
Epoch 30/30
439/439 [=====] - 51s 116ms/step - loss: 0.2683 - accuracy: 0.9023 - val_loss: 0.5720 - val_accuracy: 0.8240
```

As you can see, our model has an accuracy of ~83 % on validation data which is not bad. The accuracy can be further increased by playing around with different parameters such as

- Increasing number of Neurons
- Increasing number of hidden layers
- Increasing epochs
- Playing around with convolutional layer parameters

So this was all about how you can build your own Convolutional Neural Network for Image Classification.

If you want to learn more about CNNs, you can refer to the below links

- Yann LeCun *et al.*, 1998, [Gradient-Based Learning Applied to Document Recognition](#)
- Adit Deshpande, 2016, [The 9 Deep Learning Papers You Need To Know About \(Understanding CNNs Part 3\)](#)
- C. C. Jay Kuo, 2016, [Understanding Convolutional Neural Networks with A Mathematical Model](#)

Sign up for Top 10 Stories

By The Startup

Get smarter at building your thing. Subscribe to receive The Startup's top 10 most read stories — delivered straight into your inbox, twice a month. [Take a look](#).

Get this newsletter

Emails will be sent to infashionskin@gmail.com. Not you?

More from The Startup

Get smarter at building your thing. Follow to join The Startup's +8 million monthly readers & +756K followers.

Caio Andrade · Jul 10, 2020 ·

What Exactly Is Software Architecture?

That thing many talk about but only a few actually do — From time to time someone mentions this term. In the most diverse contexts. It's a term that has been used to express many different things, and when a word can me...

Software Architecture · 6 min read

Share your ideas with millions of readers. [Write on Medium](#)

Abdul Malik · Jul 10, 2020

Designing Data Models for Firebase

In this article we will be exploring the art of designing data models for Firebase to make it super easy to store and retrieve them. When we design a data model in our application for any entity say, a person. We build a clas...

Firebase · 4 min read

Chris Vibert · Jul 10, 2020

Software Engineering: Saying “No” Is Part of the Job

Not writing code can be just as important as writing code. — “We can deliver this feature for the client by Friday, right?” This might sound like a reasonable question for a Product Manager to ask, but for the Software...

Agile · 2 min read

Travis Kellerman · Jul 10, 2020 ·

Venture Capital Has a Language Problem

Scouts of Color can't solve it alone — The wave of calls for inclusion and change has begun to rise from the moat over the high castle walls of venture capital. In the current state, one-percent, that's 1%, of venture...

Inequality · 6 min read

Johannes Baum · Jul 10, 2020 ·

Grid-Based Movement in a Top-Down 2D RPG With Phaser 3

Classic 2D top-down RPGs often come with a movement that is grid-based. That means that your player can either walk a moment tile in the grid or no...

Game Development · 18 min read

[Read more from The Startup](#)