Analytics Vidhya

Learn Image Classification on 3 Datasets using Convolutional Neural Networks (CNN)

ALGORITHM COMPUTER VISION DATASETS IMAGE INTERMEDIATE PYTHON

Introduction

> Convolutional neural networks (CNN) – the concept behind recent breakthroughs and developments in deep learning.

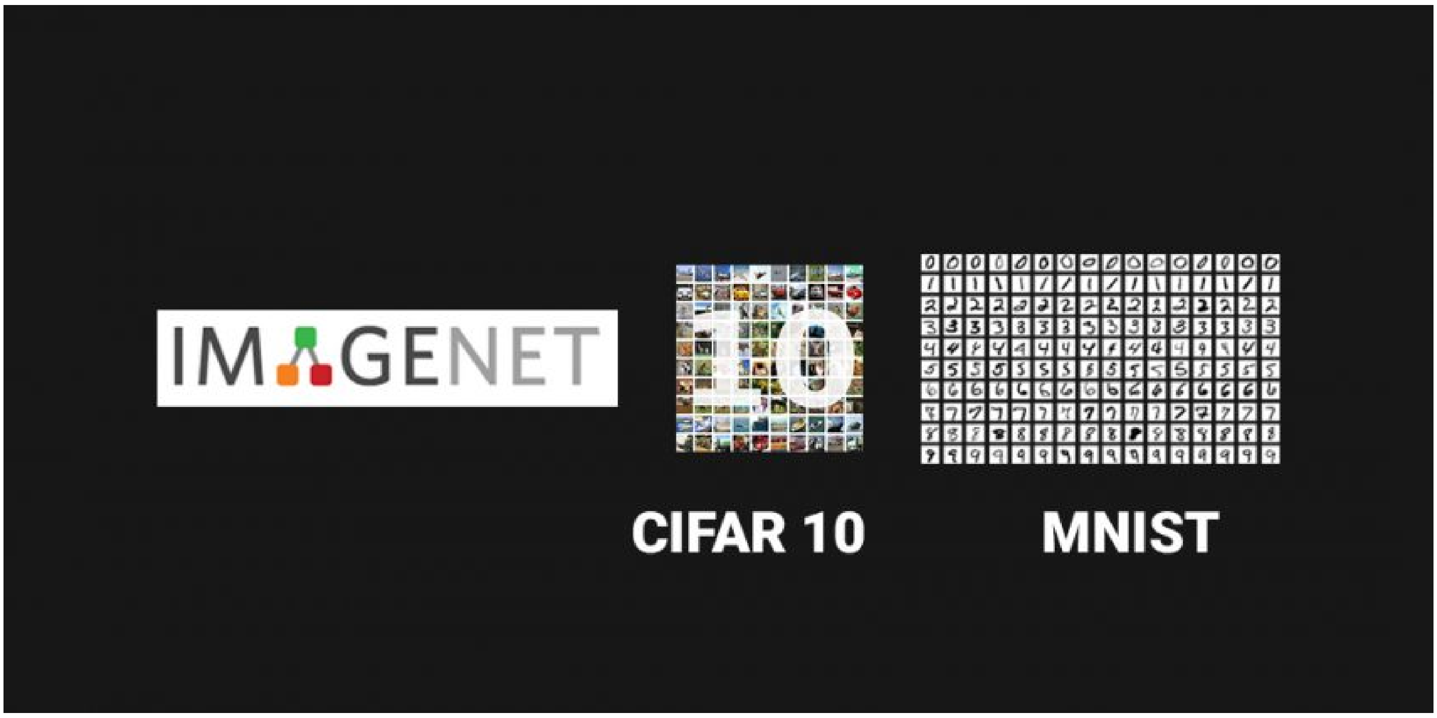
CNNs have broken the mold and ascended the throne to become the state-of-the-art computer vision technique. Among the different types of neural networks (others include recurrent neural networks (RNN), long short term memory (LSTM), artificial neural networks (ANN), etc.), CNNs are easily the most popular.

These convolutional neural network models are ubiquitous in the image data space. They work phenomenally well on computer vision tasks like image classification, object detection, image recognition, etc.

So – where can you practice your CNN skills? Well, you've come to the right place!

There are various datasets that you can leverage for applying convolutional neural networks. Here are three popular datasets:

- MNIST
- CIFAR-10
- ImageNet



In this article, we will be building image classification models using CNN on each of these datasets. That's right! We will explore MNSIT, CIFAR-10, and ImageNet to understand, in a practical manner, how CNNs work for the image classification task.

You can learn all about Convolutional Neural Networks(CNN) in this free course: Convolutional Neural Networks (CNN) from Scratch

My inspiration for writing this article is to help the community apply theoretical knowledge in a practical manner. This is a very important exercise as it not only helps you build a deeper understanding of the underlying concept but will also teach you practical details that can only be learned through implementing the concept.

If you're new to the world of neural networks, CNNs, image classification, I recommend going through these excellent in-depth tutorials:

- *Introduction to Neural Networks (Free Course!)*
- *Demystifying the Mathematics behind Convolutional Neural Networks (CNNs)*
- *Build your First Image Classification Model in just 10 Minutes*

And if you're looking to learn computer vision and deep learning in-depth, you should check out our popular course:

- *Computer Vision using Deep Learning*

Table of Contents

Note: I will be using Keras to demonstrate image classification using CNNs in this article. Keras is an excellent framework to learn when you're starting out in deep learning.

Using CNNs to Classify Hand-written Digits on MNIST Dataset

MNIST (Modified National Institute of Standards and Technology) is a well-known dataset used in Computer Vision that was built by Yann Le Cun et. al. It is composed of images that are **handwritten digits (0-9),** split into a training set of 50,000 images and a test set of 10,000 where each image is of 28 x 28 pixels in width and height.

This dataset is often used for practicing any algorithm made for image classification as the dataset is fairly easy to conquer. Hence, I recommend that this should be your first dataset if you are just foraying in the field.

MNIST comes with Keras by default and you can simply load the train and test files using a few lines of code:

```
from keras.datasets import mnist # loading the dataset (X_train, y_train), (X_test, y_test) = mnist.load_data() # let's print the shape of the dataset
```

```
print("X_train shape", X_train.shape) print("y_train shape", y_train.shape) print("X_test shape", X_test.shape) print("y_test shape", y_test.shape)
```

Here is the shape of X (features) and y (target) for the training and validation data:

```
X_train shape (60000, 28, 28) y_train shape (60000,) X_test shape (10000, 28, 28) y_test shape (10000,)
```

Before we train a CNN model, let's build a basic Fully Connected Neural Network for the dataset. The basic steps to build an image classification model using a neural network are:

1. Flatten the input image dimensions to 1D (width pixels x height pixels)
2. Normalize the image pixel values (divide by 255)
3. One-Hot Encode the categorical column
4. Build a model architecture (Sequential) with Dense layers
5. Train the model and make predictions

Here's how you can build a neural network model for MNIST. I have commented on the relevant parts of the code for better understanding:

```
# keras imports for the dataset and building our neural network
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers import Dense, Dropout, Conv2D, MaxPool2D
from keras.utils import np_utils

```

```python

# Flattening the images from the 28x28 pixels to 1D 787 pixels
X_train = X_train.reshape(60000, 784)
X_test = X_test.reshape(10000, 784)
X_train = X_train.astype('float32')
X_test = X_test.astype('float32')

# normalizing the data to help with the training
X_train /= 255
X_test /= 255

# one-hot encoding using keras' numpy-related utilities
n_classes = 10
print("Shape before one-hot encoding: ", y_train.shape)
Y_train = np_utils.to_categorical(y_train, n_classes)
Y_test = np_utils.to_categorical(y_test, n_classes)
print("Shape after one-hot encoding: ", Y_train.shape)

# building a linear stack of layers with the sequential model
model = Sequential()
# hidden layer
model.add(Dense(100, input_shape=(784,), activation='relu'))
# output layer
model.add(Dense(10, activation='softmax'))

# looking at the model summary
model.summary()
# compiling the sequential model
model.compile(loss='categorical_crossentropy', metrics=['accuracy'], optimizer='adam')
# training the model for 10 epochs
model.fit(X_train, Y_train, batch_size=128, epochs=10, validation_data=(X_test, Y_test))
```

view raw

mnist_nn.py hosted with ❤ by **GitHub**

After running the above code, you'd realized that we are getting a good validation accuracy of around 97% easily.

Let's modify the above code to build a CNN model.

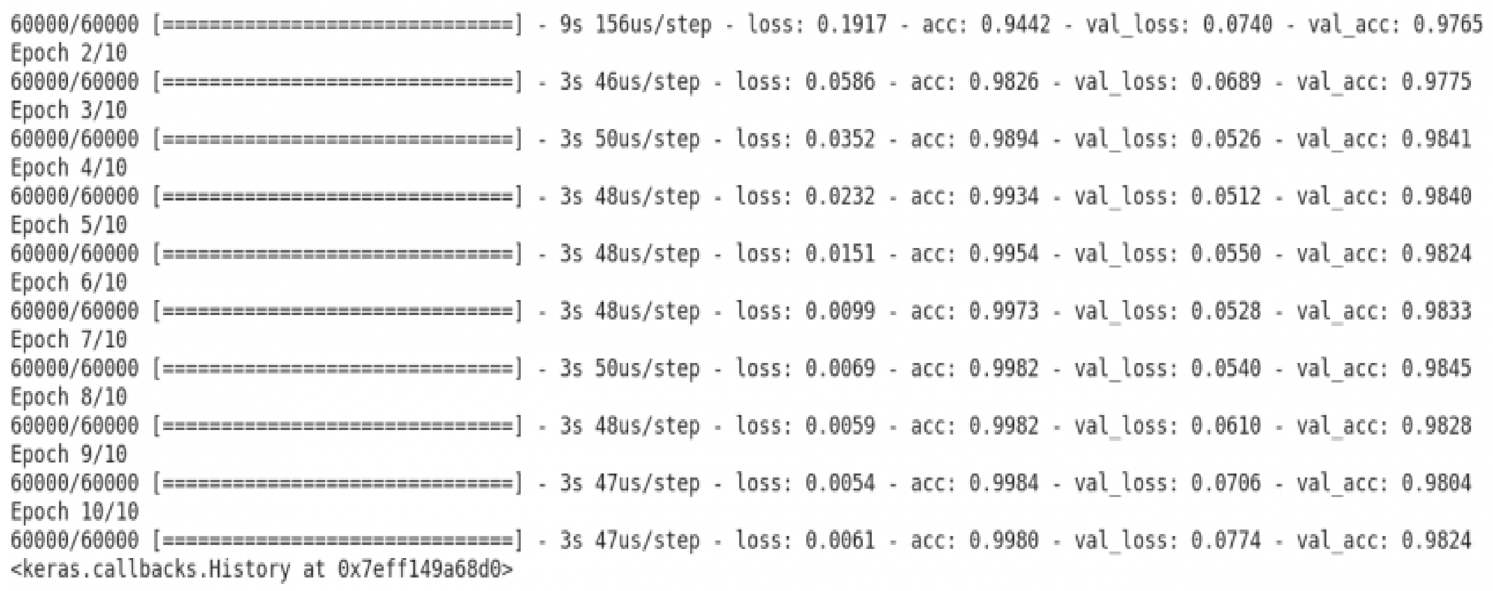
> One major advantage of using CNNs over NNs is that you do not need to flatten the input images to 1D as they are capable of working with image data in 2D. This helps in retaining the "spatial" properties of images.

Here's the full code for the CNN model:

```python
# keras imports for the dataset and building our neural network
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers import Dense, Dropout, Conv2D, MaxPool2D, Flatten
from keras.utils import np_utils

# to calculate accuracy
from sklearn.metrics import accuracy_score

# loading the dataset
(X_train, y_train), (X_test, y_test) = mnist.load_data()

# building the input vector from the 28x28 pixels
X_train = X_train.reshape(X_train.shape[0], 28, 28, 1)
X_test = X_test.reshape(X_test.shape[0], 28, 28, 1)
X_train = X_train.astype('float32')
X_test = X_test.astype('float32')

# normalizing the data to help with the training
X_train /= 255
X_test /= 255

# one-hot encoding using keras' numpy-related utilities
n_classes = 10
print("Shape before one-hot encoding: ", y_train.shape)
Y_train = np_utils.to_categorical(y_train, n_classes)
```

```python
Y_test = np_utils.to_categorical(y_test, n_classes)
print("Shape after one-hot encoding: ", Y_train.shape)

# building a linear stack of layers with the sequential model
model = Sequential()
# convolutional layer
model.add(Conv2D(25, kernel_size=(3,3), strides=(1,1), padding='valid', activation='relu', input_shape=(28,28,1)))
model.add(MaxPool2D(pool_size=(1,1)))
# flatten output of conv
model.add(Flatten())
# hidden layer
model.add(Dense(100, activation='relu'))
# output layer
model.add(Dense(10, activation='softmax'))

# compiling the sequential model
model.compile(loss='categorical_crossentropy', metrics=['accuracy'], optimizer='adam')

# training the model for 10 epochs
model.fit(X_train, Y_train, batch_size=128, epochs=10, validation_data=(X_test, Y_test))
```

Even though our max validation accuracy by using a simple neural network model was around 97%, the CNN model is able to get 98%+ with just a single convolution layer!



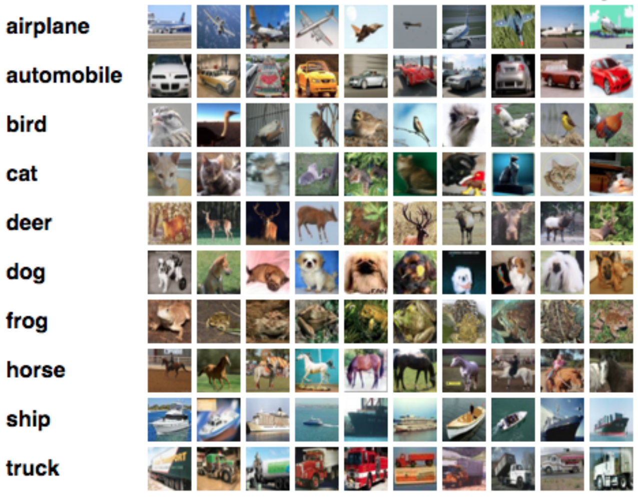
```
60000/60000 [==============================] - 9s 156us/step - loss: 0.1917 - acc: 0.9442 - val_loss: 0.0740 - val_acc: 0.9765
Epoch 2/10
60000/60000 [==============================] - 3s 46us/step - loss: 0.0586 - acc: 0.9826 - val_loss: 0.0689 - val_acc: 0.9775
Epoch 3/10
60000/60000 [==============================] - 3s 50us/step - loss: 0.0352 - acc: 0.9894 - val_loss: 0.0526 - val_acc: 0.9841
Epoch 4/10
60000/60000 [==============================] - 3s 48us/step - loss: 0.0232 - acc: 0.9934 - val_loss: 0.0512 - val_acc: 0.9840
Epoch 5/10
60000/60000 [==============================] - 3s 48us/step - loss: 0.0151 - acc: 0.9954 - val_loss: 0.0550 - val_acc: 0.9824
Epoch 6/10
60000/60000 [==============================] - 3s 48us/step - loss: 0.0099 - acc: 0.9973 - val_loss: 0.0528 - val_acc: 0.9833
Epoch 7/10
60000/60000 [==============================] - 3s 50us/step - loss: 0.0069 - acc: 0.9982 - val_loss: 0.0540 - val_acc: 0.9845
Epoch 8/10
60000/60000 [==============================] - 3s 48us/step - loss: 0.0059 - acc: 0.9982 - val_loss: 0.0610 - val_acc: 0.9828
Epoch 9/10
60000/60000 [==============================] - 3s 47us/step - loss: 0.0054 - acc: 0.9984 - val_loss: 0.0706 - val_acc: 0.9804
Epoch 10/10
60000/60000 [==============================] - 3s 47us/step - loss: 0.0061 - acc: 0.9980 - val_loss: 0.0774 - val_acc: 0.9824
<keras.callbacks.History at 0x7eff149a68d0>
```

You can go ahead and add more Conv2D layers, and also play around with the hyperparameters of the CNN model.

Identifying Images from the CIFAR-10 Dataset using CNNs

MNIST is a beginner-friendly dataset in computer vision. It's easy to score 90%+ on validation by using a CNN model. But what if you are beyond beginner and need something challenging to put your concepts to use?

That's where the CIFAR-10 dataset comes into the picture!



Here's how the developers behind CIFAR (Canadian Institute For Advanced Research) describe the dataset:

> The CIFAR-10 dataset consists of 60,000 32 x 32 colour images in 10 classes, with 6,000 images per class. There are 50,000 training images and 10,000 test images.

The important points that distinguish this dataset from MNIST are:

- Images are colored in CIFAR-10 as compared to the black and white texture of MNIST
- Each image is 32 x 32 pixel
- 50,000 training images and 10,000 testing images

Now, these images are taken in varying lighting conditions and at different angles, and since these are colored images, you will see that there are many variations in the color itself of similar objects (for example, the color of ocean water). If you use the simple CNN architecture that we saw in the MNIST example above, you will get a low validation accuracy of around 60%.

That's a key reason why I recommend CIFAR-10 as a good dataset to practice your hyperparameter tuning skills for CNNs. The good thing is that just like MNIST, CIFAR-10 is also easily available in Keras.

You can simply load the dataset using the following code:

```
from keras.datasets import cifar10 # loading the dataset (X_train, y_train), (X_test, y_test) = cifar10.load_data()
```

Here's how you can build a decent (around 78-80% on validation) CNN model for CIFAR-10. Notice how the shape values have been updated from (28, 28, 1) to (32, 32, 3) according to the size of the images:

```
# keras imports for the dataset and building our neural network
from keras.datasets import cifar10
from keras.models import Sequential
from keras.layers import Dense, Dropout, Conv2D, MaxPool2D, Flatten
from keras.utils import np_utils

# loading the dataset
(X_train, y_train), (X_test, y_test) = cifar10.load_data()

# # building the input vector from the 32x32 pixels
X_train = X_train.reshape(X_train.shape[0], 32, 32, 3)
X_test = X_test.reshape(X_test.shape[0], 32, 32, 3)
X_train = X_train.astype('float32')
X_test = X_test.astype('float32')

# normalizing the data to help with the training
X_train /= 255
```

```python
X_test /= 255

# one-hot encoding using keras' numpy-related utilities
n_classes = 10
print("Shape before one-hot encoding: ", y_train.shape)
Y_train = np_utils.to_categorical(y_train, n_classes)
Y_test = np_utils.to_categorical(y_test, n_classes)
print("Shape after one-hot encoding: ", Y_train.shape)

# building a linear stack of layers with the sequential model
model = Sequential()

# convolutional layer
model.add(Conv2D(50, kernel_size=(3,3), strides=(1,1), padding='same', activation='relu', input_shape=(32, 32, 3)))

# convolutional layer
model.add(Conv2D(75, kernel_size=(3,3), strides=(1,1), padding='same', activation='relu'))
model.add(MaxPool2D(pool_size=(2,2)))
model.add(Dropout(0.25))

model.add(Conv2D(125, kernel_size=(3,3), strides=(1,1), padding='same', activation='relu'))
model.add(MaxPool2D(pool_size=(2,2)))
model.add(Dropout(0.25))

# flatten output of conv
model.add(Flatten())

# hidden layer
model.add(Dense(500, activation='relu'))
model.add(Dropout(0.4))
model.add(Dense(250, activation='relu'))
model.add(Dropout(0.3))
# output layer
model.add(Dense(10, activation='softmax'))

# compiling the sequential model
model.compile(loss='categorical_crossentropy', metrics=['accuracy'], optimizer='adam')

# training the model for 10 epochs
model.fit(X_train, Y_train, batch_size=128, epochs=10, validation_data=(X_test, Y_test))
```

view raw

cnn_cifar10.py hosted with ❤ by **GitHub**

Here's what I changed in the model:

- Increased the number of Conv2D layers to build a deeper model
- Increased number of filters to learn more features
- Added Dropout for regularization
- Added more Dense layers

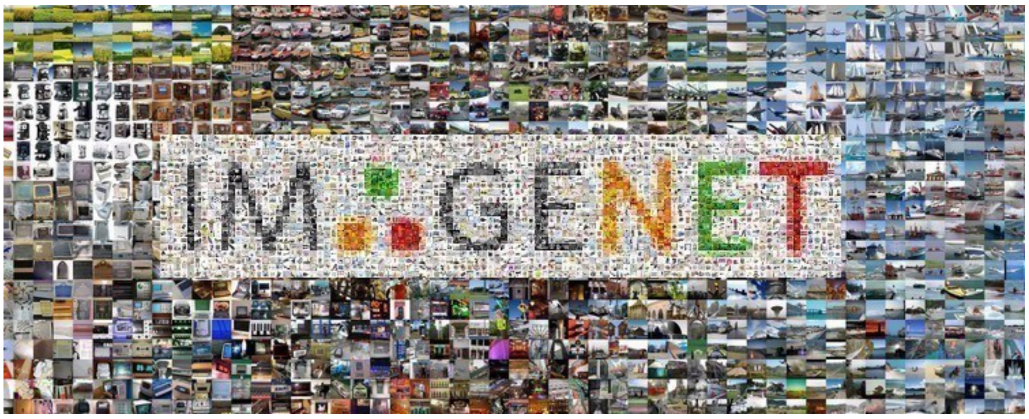
Training and validation accuracy across epochs:

```
Train on 50000 samples, validate on 10000 samples
Epoch 1/10
50000/50000 [==============================] - 12s 240us/step - loss: 1.5801 - acc: 0.4223 - val_loss: 1.1511 - val_acc: 0.5900
Epoch 2/10
50000/50000 [==============================] - 10s 205us/step - loss: 1.1047 - acc: 0.6080 - val_loss: 0.9788 - val_acc: 0.6579
Epoch 3/10
50000/50000 [==============================] - 10s 206us/step - loss: 0.9142 - acc: 0.6796 - val_loss: 0.8076 - val_acc: 0.7222
Epoch 4/10
50000/50000 [==============================] - 10s 206us/step - loss: 0.8034 - acc: 0.7193 - val_loss: 0.7635 - val_acc: 0.7367
Epoch 5/10
50000/50000 [==============================] - 10s 205us/step - loss: 0.7203 - acc: 0.7488 - val_loss: 0.7122 - val_acc: 0.7547
Epoch 6/10
50000/50000 [==============================] - 10s 206us/step - loss: 0.6551 - acc: 0.7716 - val_loss: 0.6956 - val_acc: 0.7573
Epoch 7/10
50000/50000 [==============================] - 10s 208us/step - loss: 0.5928 - acc: 0.7936 - val_loss: 0.6649 - val_acc: 0.7719
Epoch 8/10
50000/50000 [==============================] - 10s 207us/step - loss: 0.5388 - acc: 0.8095 - val_loss: 0.6746 - val_acc: 0.7718
Epoch 9/10
50000/50000 [==============================] - 10s 207us/step - loss: 0.5023 - acc: 0.8231 - val_loss: 0.6541 - val_acc: 0.7786
Epoch 10/10
50000/50000 [==============================] - 10s 208us/step - loss: 0.4657 - acc: 0.8346 - val_loss: 0.6460 - val_acc: 0.7832
<keras.callbacks.History at 0x7efeec8aa198>
```

You can easily eclipse this performance by tuning the above model. Once you have mastered CIFAR-10, there's also CIFAR-100 available in Keras that you can use for further practice. Since it has 100 classes, it won't be an easy task to achieve!

Categorizing the Images of ImageNet using CNNs

Now that you have mastered MNIST and CIFAR-10, let's take this problem a notch higher. Here, we will take a look at the famous ImageNet dataset.



ImageNet is the main database behind the **ImageNet Large Scale Recognition Challenge (ILSVRC)**. This is like the Olympics of Computer Vision. This is the competition that made CNNs popular the first time and every year, the best research teams across industries and academia compete with their best algorithms on computer vision tasks.

About the ImageNet Dataset

> The ImageNet dataset has more than 14 million images, hand-labeled across 20,000 categories.

Also, unlike the MNIST and CIFAR-10 datasets that we have already discussed, the images in ImageNet are of decent resolution (224 x 224) and that's what poses a challenge for us: 14 million images, each 224 by 224 pixels. Processing a dataset of this size requires a great amount of computing power in terms of CPU, GPU, and RAM.

The downside – that might be too much for an everyday laptop. So what's the alternative solution? How can an enthusiast work with the ImageNet dataset?

That's where Fast.ai's Imagenette dataset comes in

Imagenette is a dataset that's extracted from the large ImageNet collection of images. The reason behind releasing Imagenette is that researchers and students can practice on ImageNet level images without needing that much compute resources.

In the words of Jeremy Howard himself:

> "I (Jeremy Howard, that is) mainly made Imagenette because I wanted a small vision dataset I could use to quickly see if my algorithm ideas might have a chance of working. They normally don't, but testing them on Imagenet takes a really long time for me to find that out, especially because I'm interested in algorithms that perform particularly well at the *end* of training.
>
> But I think this can be a useful dataset for others as well."

And that's what we will also use for practicing!

1. Download the Imagenette Dataset

Here's how you can fetch the dataset (commands for your terminal):

```
$ wget https://s3.amazonaws.com/fast-ai-imageclas/imagenette2.tgz $ tar -xf imagenette2.tgz
```

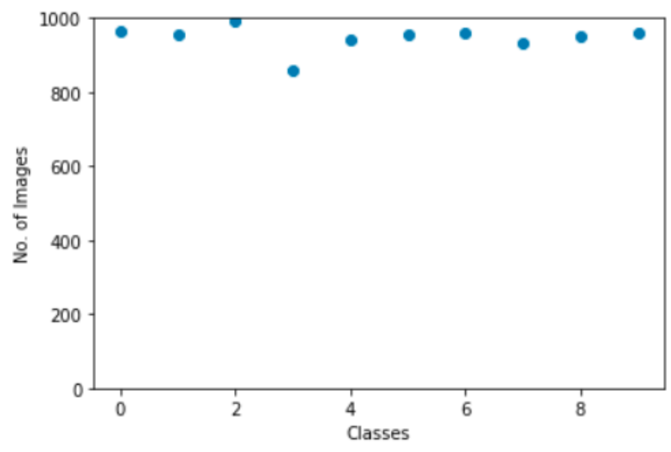
Once you have downloaded the dataset, you will notice that it has two folders – "train" and "val". These contain the training and validation set respectively. Inside each folder, there are separate folders for each class. Here's the mapping of the classes:

```python
imagenette_map = {
    "n01440764" : "tench",
    "n02102040" : "springer",
    "n02979186" : "casette_player",
    "n03000684" : "chain_saw",
    "n03028079" : "church",
    "n03394916" : "French_horn",
    "n03417042" : "garbage_truck",
    "n03425413" : "gas_pump",
    "n03445777" : "golf_ball",
    "n03888257" : "parachute"
}
```

view raw

imagenette_map.py hosted with ❤ by **GitHub**

These classes have the same ID in the original ImageNet dataset. Each of the classes has approximately 1000 images so overall, it's a balanced dataset.



2. Loading Images using ImageDataGenerator

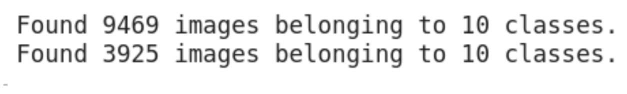
Keras has this useful functionality for loading large images (like we have here) without maxing out the RAM, by doing it in small batches. ImageDataGenerator in combination with fit_generator provides this functionality:

```
from keras.preprocessing.image import ImageDataGenerator

# create a new generator
imagegen = ImageDataGenerator()
# load train data
train = imagegen.flow_from_directory("imagenette2/train/", class_mode="categorical", shuffle=False, batch_size=128, target_size=(
# load val data
val = imagegen.flow_from_directory("imagenette2/val/", class_mode="categorical", shuffle=False, batch_size=128, target_size=(224,
```

view raw

datagen.py hosted with ❤ by **GitHub**

The ImageDataGenerator itself inferences the class labels and the number of classes from the folder names.



```
Found 9469 images belonging to 10 classes.
Found 3925 images belonging to 10 classes.
```

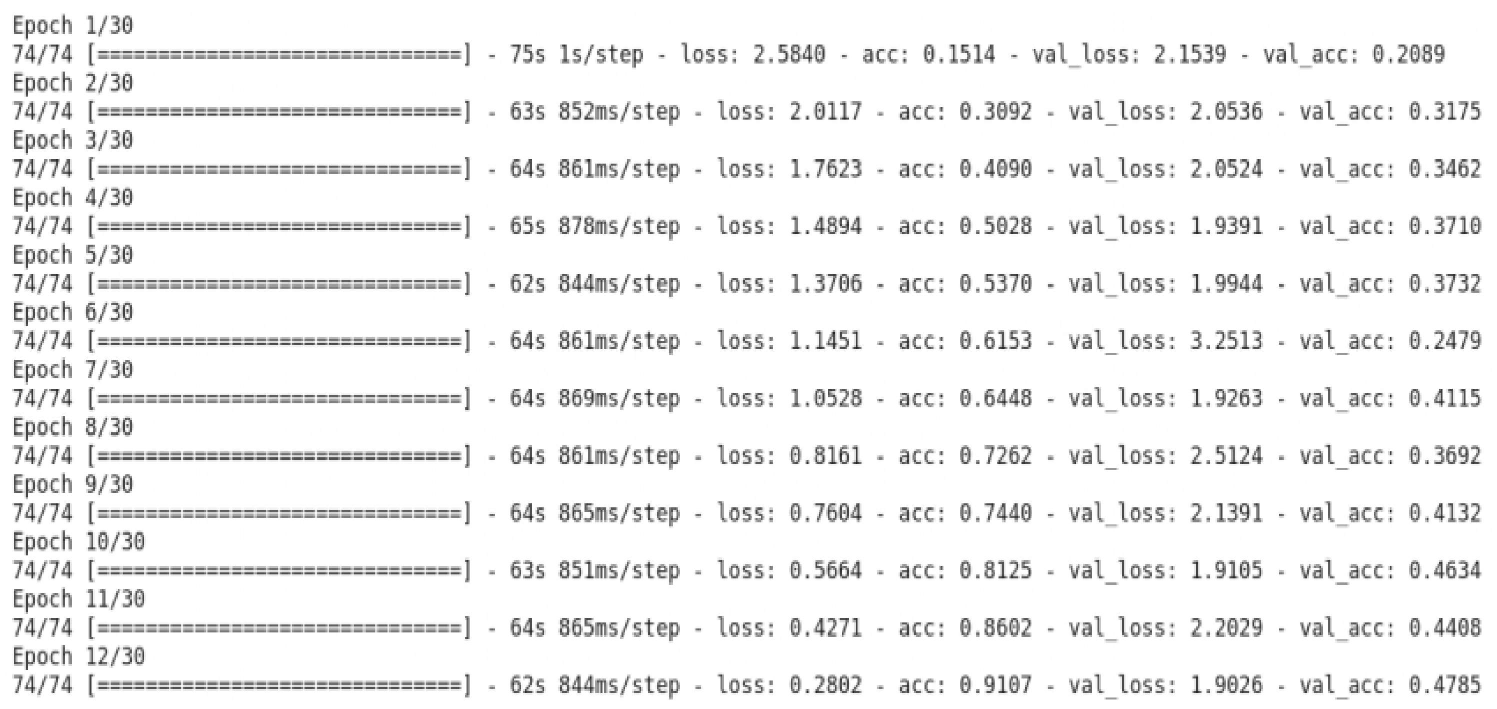
3. Building a Basic CNN model for Image Classification

Let’s build a basic CNN model for our Imagenette dataset (for the purpose of image classification):

```
from keras.models import Sequential
from keras.layers import Conv2D, MaxPool2D, Flatten, Dense, InputLayer, BatchNormalization, Dropout

# build a sequential model
model = Sequential()
model.add(InputLayer(input_shape=(224, 224, 3)))

# 1st conv block
model.add(Conv2D(25, (5, 5), activation='relu', strides=(1, 1), padding='same'))
model.add(MaxPool2D(pool_size=(2, 2), padding='same'))
# 2nd conv block
model.add(Conv2D(50, (5, 5), activation='relu', strides=(2, 2), padding='same'))
model.add(MaxPool2D(pool_size=(2, 2), padding='same'))
model.add(BatchNormalization())
```

```python
# 3rd conv block
model.add(Conv2D(70, (3, 3), activation='relu', strides=(2, 2), padding='same'))
model.add(MaxPool2D(pool_size=(2, 2), padding='valid'))
model.add(BatchNormalization())
# ANN block
model.add(Flatten())
model.add(Dense(units=100, activation='relu'))
model.add(Dense(units=100, activation='relu'))
model.add(Dropout(0.25))
# output layer
model.add(Dense(units=10, activation='softmax'))

# compile model
model.compile(loss='categorical_crossentropy', optimizer="adam", metrics=['accuracy'])
# fit on data for 30 epochs
model.fit_generator(train, epochs=30, validation_data=val)
```

When we compare the validation accuracy of the above model, you'll realize that even though it is a more deep architecture than what we have utilized so far, we are only able to get a validation accuracy of around 40-50%.



```
Epoch 1/30
74/74 [==============================] - 75s 1s/step - loss: 2.5840 - acc: 0.1514 - val_loss: 2.1539 - val_acc: 0.2089
Epoch 2/30
74/74 [==============================] - 63s 852ms/step - loss: 2.0117 - acc: 0.3092 - val_loss: 2.0536 - val_acc: 0.3175
Epoch 3/30
74/74 [==============================] - 64s 861ms/step - loss: 1.7623 - acc: 0.4090 - val_loss: 2.0524 - val_acc: 0.3462
Epoch 4/30
74/74 [==============================] - 65s 878ms/step - loss: 1.4894 - acc: 0.5028 - val_loss: 1.9391 - val_acc: 0.3710
Epoch 5/30
74/74 [==============================] - 62s 844ms/step - loss: 1.3706 - acc: 0.5370 - val_loss: 1.9944 - val_acc: 0.3732
Epoch 6/30
74/74 [==============================] - 64s 861ms/step - loss: 1.1451 - acc: 0.6153 - val_loss: 3.2513 - val_acc: 0.2479
Epoch 7/30
74/74 [==============================] - 64s 869ms/step - loss: 1.0528 - acc: 0.6448 - val_loss: 1.9263 - val_acc: 0.4115
Epoch 8/30
74/74 [==============================] - 64s 861ms/step - loss: 0.8161 - acc: 0.7262 - val_loss: 2.5124 - val_acc: 0.3692
Epoch 9/30
74/74 [==============================] - 64s 865ms/step - loss: 0.7604 - acc: 0.7440 - val_loss: 2.1391 - val_acc: 0.4132
Epoch 10/30
74/74 [==============================] - 63s 851ms/step - loss: 0.5664 - acc: 0.8125 - val_loss: 1.9105 - val_acc: 0.4634
Epoch 11/30
74/74 [==============================] - 64s 865ms/step - loss: 0.4271 - acc: 0.8602 - val_loss: 2.2029 - val_acc: 0.4408
Epoch 12/30
74/74 [==============================] - 62s 844ms/step - loss: 0.2802 - acc: 0.9107 - val_loss: 1.9026 - val_acc: 0.4785
```

There can be many reasons for this, such as our model is not complex enough to learn the underlying patterns of images, or maybe the training data is too small to accurately generalize across classes.

Step up – transfer learning.

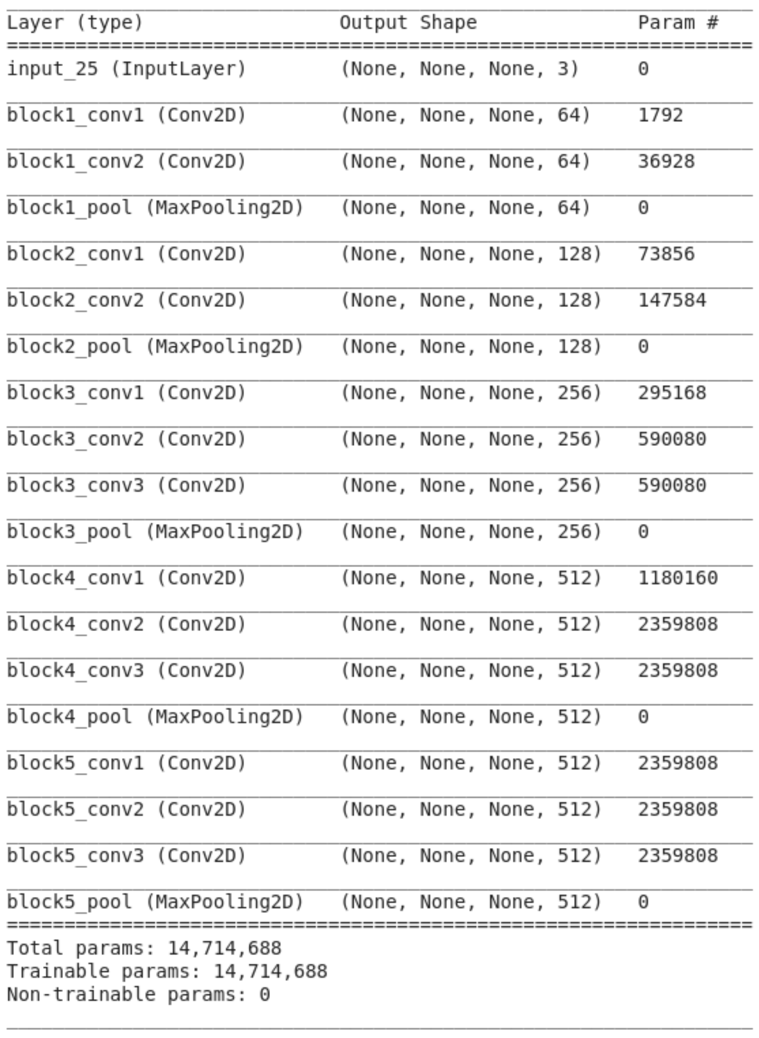
4. Using Transfer Learning (VGG16) to improve accuracy

VGG16 is a CNN architecture that was the first runner-up in the 2014 ImageNet Challenge. It's designed by the Visual Graphics Group at Oxford and has 16 layers in total, with 13 convolutional layers themselves. We will load the pre-trained weights of this model so that we can utilize the useful features this model has learned for our task.

Downloading weights of VGG16

```
from keras.applications import VGG16 # include top should be False to remove the softmax layer
pretrained_model = VGG16(include_top=False, weights='imagenet') pretrained_model.summary()
```

Here’s the architecture of the model:



```
Layer (type)                 Output Shape              Param #
=================================================================
input_25 (InputLayer)        (None, None, None, 3)     0
_________________________________________________________________
block1_conv1 (Conv2D)        (None, None, None, 64)    1792
_________________________________________________________________
block1_conv2 (Conv2D)        (None, None, None, 64)    36928
_________________________________________________________________
block1_pool (MaxPooling2D)   (None, None, None, 64)    0
_________________________________________________________________
block2_conv1 (Conv2D)        (None, None, None, 128)   73856
_________________________________________________________________
block2_conv2 (Conv2D)        (None, None, None, 128)   147584
_________________________________________________________________
block2_pool (MaxPooling2D)   (None, None, None, 128)   0
_________________________________________________________________
block3_conv1 (Conv2D)        (None, None, None, 256)   295168
_________________________________________________________________
block3_conv2 (Conv2D)        (None, None, None, 256)   590080
_________________________________________________________________
block3_conv3 (Conv2D)        (None, None, None, 256)   590080
_________________________________________________________________
block3_pool (MaxPooling2D)   (None, None, None, 256)   0
_________________________________________________________________
block4_conv1 (Conv2D)        (None, None, None, 512)   1180160
_________________________________________________________________
block4_conv2 (Conv2D)        (None, None, None, 512)   2359808
_________________________________________________________________
block4_conv3 (Conv2D)        (None, None, None, 512)   2359808
_________________________________________________________________
block4_pool (MaxPooling2D)   (None, None, None, 512)   0
_________________________________________________________________
block5_conv1 (Conv2D)        (None, None, None, 512)   2359808
_________________________________________________________________
block5_conv2 (Conv2D)        (None, None, None, 512)   2359808
_________________________________________________________________
block5_conv3 (Conv2D)        (None, None, None, 512)   2359808
_________________________________________________________________
block5_pool (MaxPooling2D)   (None, None, None, 512)   0
=================================================================
Total params: 14,714,688
Trainable params: 14,714,688
Non-trainable params: 0
_________________________________________________________________
```

Generate features from VGG16

Let’s extract useful features that VGG16 already knows from our dataset’s images:

```
from keras.utils import to_categorical # extract train and val features vgg_features_train =
pretrained_model.predict(train) vgg_features_val = pretrained_model.predict(val)
```

```
# OHE target column train_target = to_categorical(train.labels) val_target = to_categorical(val.labels)
```

Once the above features are ready, we can just use them to train a basic Fully Connected Neural Network in Keras:

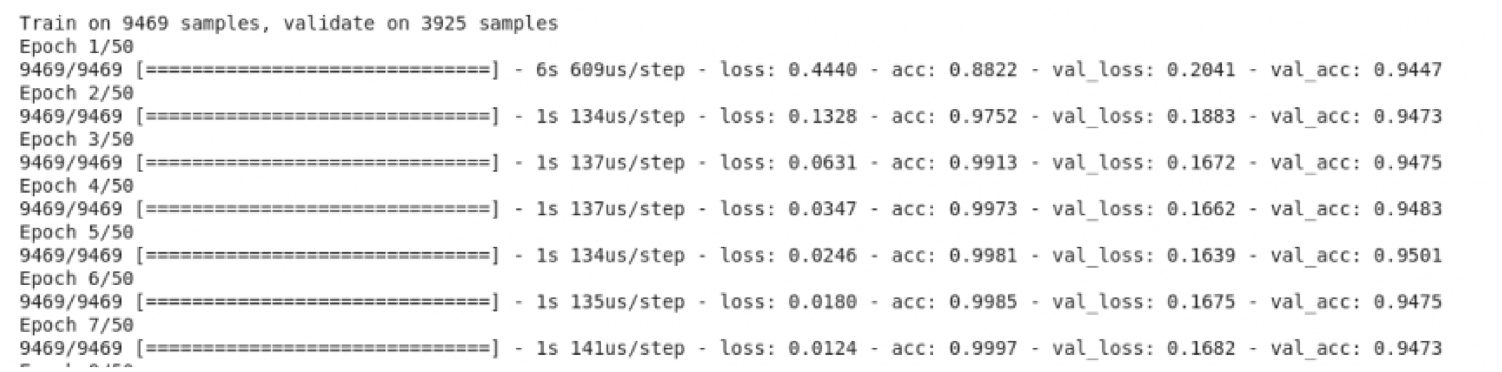
```
model2 = Sequential()
model2.add(Flatten(input_shape=(7,7,512)))
model2.add(Dense(100, activation='relu'))
model2.add(Dropout(0.5))
model2.add(BatchNormalization())
model2.add(Dense(10, activation='softmax'))

```

```python
# compile the model
model2.compile(optimizer='adam', metrics=['accuracy'], loss='categorical_crossentropy')

model2.summary()

# train model using features generated from VGG16 model
model2.fit(vgg_features_train, train_target, epochs=50, batch_size=128, validation_data=(vgg_features_val, val_target))
```

view raw

fcnn_vgg.py hosted with ❤ by **GitHub**

Notice how quickly your model starts converging. In just 10 epochs, you have a 94%+ validation accuracy. Isn't that amazing?



```
Train on 9469 samples, validate on 3925 samples
Epoch 1/50
9469/9469 [==============================] - 6s 609us/step - loss: 0.4440 - acc: 0.8822 - val_loss: 0.2041 - val_acc: 0.9447
Epoch 2/50
9469/9469 [==============================] - 1s 134us/step - loss: 0.1328 - acc: 0.9752 - val_loss: 0.1883 - val_acc: 0.9473
Epoch 3/50
9469/9469 [==============================] - 1s 137us/step - loss: 0.0631 - acc: 0.9913 - val_loss: 0.1672 - val_acc: 0.9475
Epoch 4/50
9469/9469 [==============================] - 1s 137us/step - loss: 0.0347 - acc: 0.9973 - val_loss: 0.1662 - val_acc: 0.9483
Epoch 5/50
9469/9469 [==============================] - 1s 134us/step - loss: 0.0246 - acc: 0.9981 - val_loss: 0.1639 - val_acc: 0.9501
Epoch 6/50
9469/9469 [==============================] - 1s 135us/step - loss: 0.0180 - acc: 0.9985 - val_loss: 0.1675 - val_acc: 0.9475
Epoch 7/50
9469/9469 [==============================] - 1s 141us/step - loss: 0.0124 - acc: 0.9997 - val_loss: 0.1682 - val_acc: 0.9473
```

In case you have mastered the Imagenette dataset, fastai has also released two variants which include classes you'll find difficult to classify:

- Imagewoof: 10 classes of dog breeds, a more difficult problem to classify
- Image□ ("wang"): A combination of Imagenette and Imagewoof and a couple of tricks that make it a harder problem

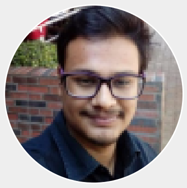
Where to go from here?

Apart from the datasets we've above, you can also use the below datasets for building computer vision algorithms. In fact, consider this a challenge. Can you apply your CNN knowledge to beat the benchmark score on these datasets?

- Fashion MNIST – MNIST-like dataset of clothes and apparel. Instead of digits, the images show a type of apparel (T-shirt, trousers, bag, etc.)
- Caltech 101 – Another challenging dataset that I found for image classification

I also suggest that before going for transfer learning, try improving your base CNN models. You can learn from the architectures of VGG16, ZFNet, etc. for some clues on hyperparameter tuning and you can use the same *ImageDataGenerator* to augment your images and increase the size of the dataset.

Article Url - https://www.analyticsvidhya.com/blog/2020/02/learn-image-classification-cnn-convolutional-neural-networks-3-datasets/



Mohd Sanad Zaki Rizvi

A computer science graduate, I have previously worked as a Research Assistant at the University of Southern California(USC-ICT) where I employed NLP and ML to make better virtual STEM mentors. My

research interests include using AI and its allied fields of NLP and Computer Vision for tackling real-world problems.