Nanonets

DEEP LEARNING CLASSIFICATION GETTING STARTED

TUTORIAL TENSORFLOW FOOD CLASSIFICATION

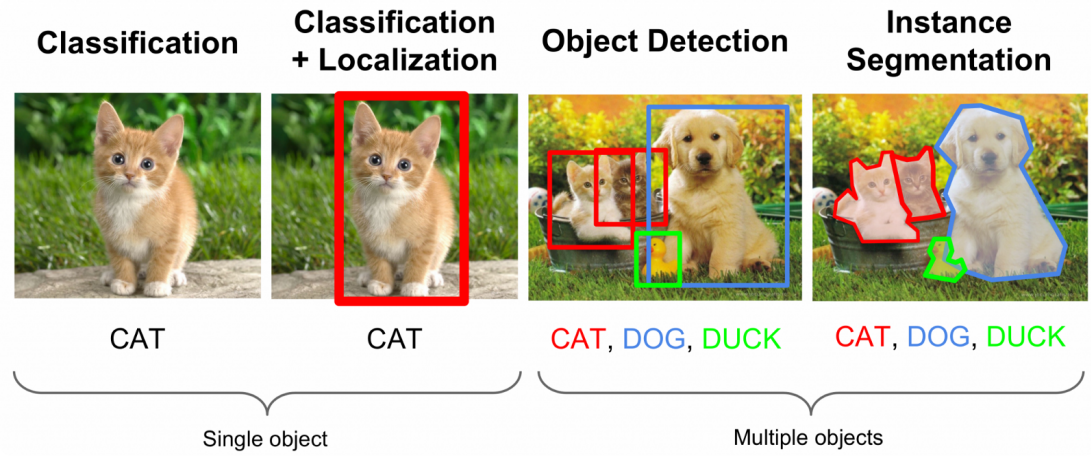
How To Easily Classify Food Using Deep Learning and Tensorflow

by **Bharath Raj** a year ago 10 MIN READ

> ## An in-depth tutorial on creating Deep Learning models for Multi Label

Nanonets

By now you would have heard about Convolutional Neural Networks (CNNs) and its efficacy in classifying images. The accuracy of CNNs in image classification is quite remarkable and its real-life applications through APIs quite profound.



Examples of Classification, Localization, Object Detection and Instance Segmentation. (Source)

But sometimes, this technique may not be adequate. An image may represent multiple attributes. For instance, all of the following tags are valid for the below image. A simple classifier would get confused on what label to provide in such a scenario.

Nanonets

An image with multiple possible correct labels. (Source)

This problem is known as **Multi-Label classification**.

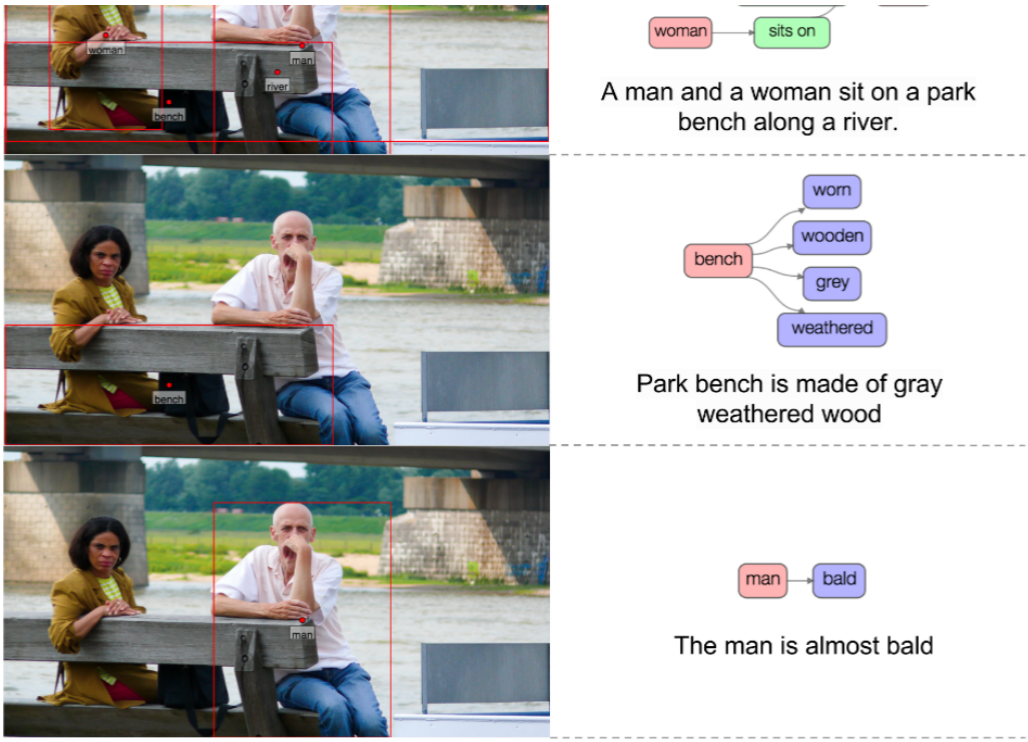
Why Multi-Label Classification ?

There are many applications where assigning multiple attributes to an image is necessary. In fact, it is more natural to think of images as belonging to multiple classes rather than a single class. Below are some applications of Multi Label Classification.

1. Scene Understanding

Multi Label Classification provides an easy to calculate prior for complex Scene Understanding algorithms. Identifying various possible tags for an image can help the Scene Understanding algorithm to create multiple vivid descriptions for the image.

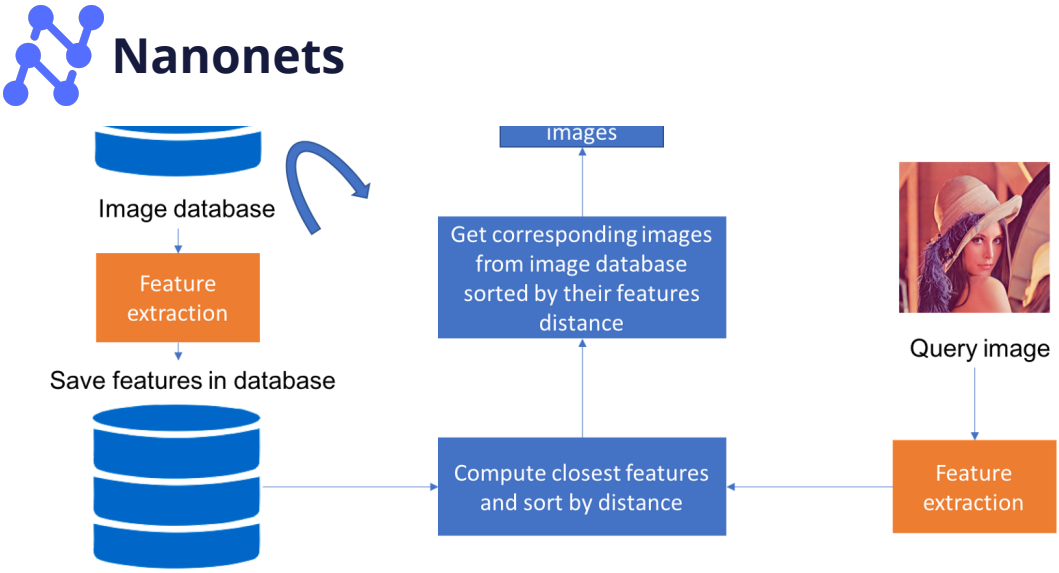
Nanonets



Multiple descriptions can be created for a scene based on the labels identified from the image. (Source)

2. Content-Based Retrieval

Multi Label tags can enhance the ability of search engines to retrieve very specific queries of a given product. For instance, we could provide multiple tags for an image of a fashion model wearing branded attire. A search engine can retrieve this result when you search for any one of the tags. A Multi Label Classification engine can automatically build up a database for the search engine.



Content Based Image Retrieval in action. The Multi Label classifier performs the function of the Feature Extraction module in the above flowchart. (Source)

Moreover, we can use the tags to recommend related products based on the user's activity or preferences. For instance, you can recommend similar songs or movies based on the user's activity. A Multi Label Classifier can be used to automatically index such songs and movies.

How does label classification works?

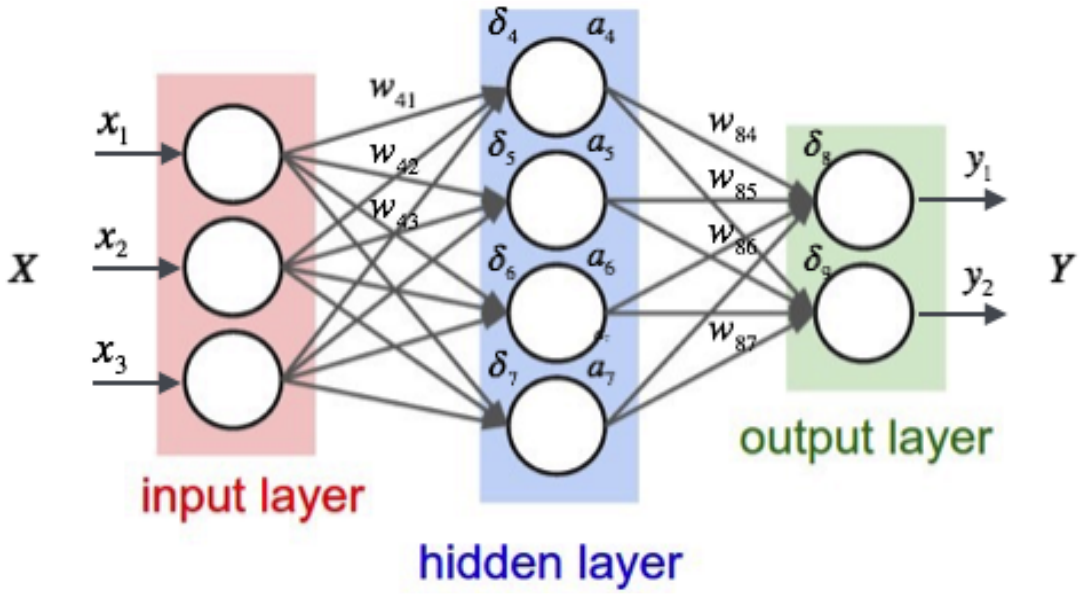
If you are familiar with Machine Learning algorithms for classification, some minor modifications are enough to make the same algorithm work for a multi label problem. In any case, let us do a small review of how classification works, and how it can be expanded to a multi label scenario. For the rest of this blog, we will focus on implementing the same for images.

Single Label Classification

Neural Networks are among the most powerful (and popular) algorithms used for classification. They take inputs

Nanonets

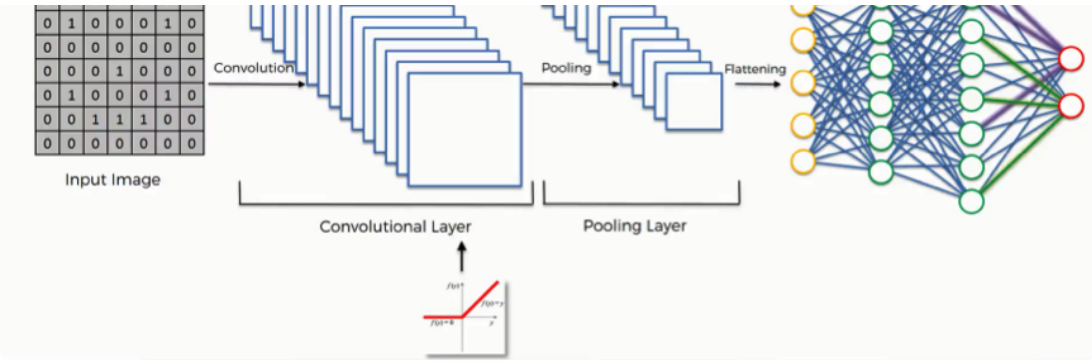
compared with the ground truth labels and the computation process is tweaked (i.e. trained) to yield better results. To train the Neural Network, we feed our input data in the form of feature vectors that represent the important gist of the data.



A Multi Layer Perceptron. (Source)

One hurdle you might have noticed is the issue of encoding images into a feature vector. Convolutional Neural Networks (CNNs) are used for this purpose. Convolutions extract important features from the images and convert them into a vector representation for further processing. The rest of the processing in a CNN is similar to that of a Multi Layered Perceptron. This, in a nutshell, is how single label classification is performed.

Nanonets



A Convolutional Neural Network. (Source)

Multi Label Classification

Now, how do we adapt this model for Multi Label Classification ? There are several strategies for doing the same.

Method 1—Problem Transformation

In this case, we will transform the Multi Label problem into a Multi Class problem. One way of doing this is by training a separate classifier for each label. This method has the obvious downside of training too many classifiers. This also ignores possible correlation between each label.

Another method is by encoding each possible combination of labels as a separate class, thereby creating a label powerset. This method works well for a small number of label combinations, but they are hard to scale for large number of label combinations. For just 10 labels, we would have get a power set of size 1024 (2 raised to the power 10)!

Method 2—Adapting the Algorithm

Sometimes, making some minor modifications to the algorithm would be enough for tackling a Multi Label

Sigmoid layer and then use Binary Cross Entropy to optimize the model.

Clearly there are a lot of strategies that can be explored. Often, one strategy may not work best for all kinds of data and hence requires lots of experimentation.

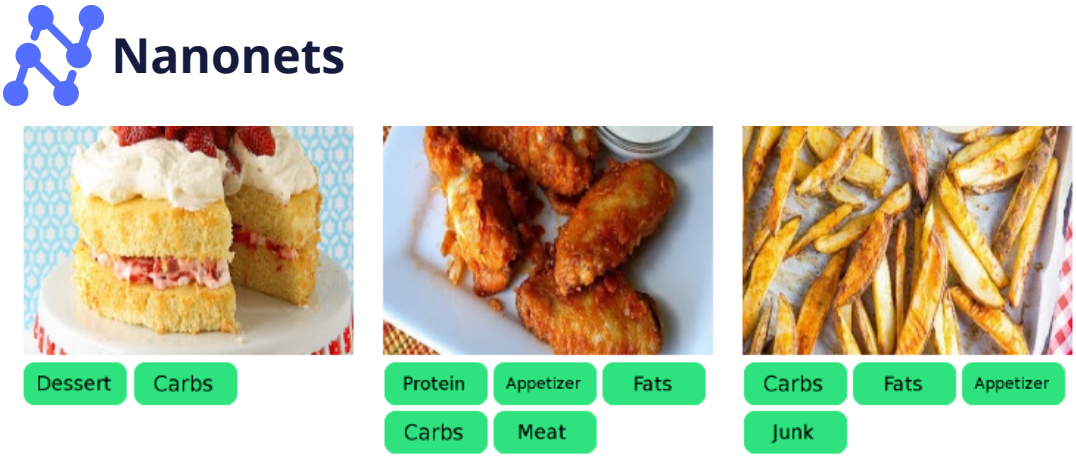
Multi Label Food Classification

The theory sounds alright but, how do we implement it? In this section, we will build our own Multi Label Food Classification algorithm using Keras (with TensorFlow backend). We will modify a simple CNN model to enable multi label classification. We will then do a comparison with Nanonets Multi Label Classification API.

All the code is available on GitHub over here. You can follow the GitHub repository for an in-depth guide to replicate the experiments.

Problem Description

Let us work on a possible real life application of Multi Label Classification. Given a food item, we would like to identify possible tags for the image. For instance, given an image of a cake, we would like our model to provide tags such as “carbs” and “dessert”.



Sample images and their respective tags.

Such a model is extremely useful for Content Based Retrieval for businesses based on the food industry. For instance, we can create an automated dietary planar app based on the requirements of the user and retrieve relevant images and recipes for the appropriate food items.

Part 1—Data Collection

The first step is to collect and clean the data. I sampled around 2000 images from the Recipes5k dataset and resized them to size 224 x 224. The original dataset had annotations of the ingredients of a food item. However, there were more than 1000 possible ingredients (i.e. labels) and this would created highly sparse label vectors. Hence, I created my own set of annotations for the same images.

In our case, an image can have at most 10 possible labels. The list of labels are: [“Soups”, “Mains”, “Appetizer”, “Dessert” ,“Protein”, “Fats”, “Carbs”, “Healthy”, “Junk”, “Meat”]. To encode the labels in a format that can be utilized by the neural network, we create a 10 dimensional vector such that there is a “1” if a label is present in the image and “0” if a label is absent.

Nanonets

Carbs". This greatly simplified the annotation process, and I wrote a simple python function to carry most of the heavy lifting. While this strategy makes the process simpler, it may create some noisy annotations (i.e. slightly wrong) and could impact the final accuracy. Nevertheless, for this toy experiment, we proceed as such. A sample annotation for a cake image and its label is shown below.

A sample image and its vector format.

Clearly, we are restricted by the quantity of data in hand. To better enhance the training ability of our CNN, we can perform Data Augmentation and Transfer Learning.

Part 2—Building the Model

We will define the model using Keras as follows. The below model is a pretrained ResNet-50 with two Dense layers in the end. Notice that we used a `sigmoid` activation rather than `softmax` for the Dense Layer.

Nanonets

```
img = Input(shape = (224, 224, 3))
model = ResNet50(
    weights = 'imagenet',include_top = False,
    input_tensor = img, input_shape = None, pooling = 'avg')
final_layer = model.layers[-1].output
dense_layer_1 = Dense(128, activation = 'relu')(final_layer)
output_layer = Dense(10, activation = 'sigmoid')(dense_layer
model = Model(input = img, output = output_layer)
# We will use a Binary Cross Entropy loss function. To calcu
# the accuracy of the model, we use the F1 score averaged by
# samples (or similar) as the metric.
model.compile(
optimizer = 'adam',metrics = ['accuracy'], loss = 'binary_cr
```

Part 3—Training

The data was split into train, validation and test sets. The data is normalized channel-wise before being fed into the CNN. Since our dataset is relatively small, we can directly use `model.fit()` to train our model. This is shown is the following code snippet:

```
model.fit(
trainX, trainY, batch_size = 32, epochs = 50,
validation_data = (valX, valY))
```

Part 4—Inference

Now that we have a trained model, we can visualize its performance using `model.predict()` . This will output an array with each element representing the probability of a tag (or label). We can obtain a binary vector by rounding the predicted array such that a `1` signifies the presence of a tag

Nanonets

image below.

Categories:
{0:'Healthy', 1:'Junk', 2:'Dessert', 3:'Appetizer', 4:'Mains', 5:'Soups', 6:'Carbs', 7:'Protein', 8:'Fats', 9:'Meat'}

Prediction:
[1, 0, 0, 0, 1, 0, 0, 1, 0, 0] --> Output array
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9] --> Index of each element
[0, 4, 7] --> Index of each element where output is 1
['Healthy', 'Mains', 'Protein'] --> Corresponding category

Decoding the output from the Neural Network

To analyze the performance, we repeat the experiment with different models pre-trained on the ImageNet dataset. Overall, the following pre-trained models were used:

- ResNet-50
- DenseNet-121
- Xception
- MobileNet

Check out this dedicated Github repo to have a look at the complete code.

That was great! But...

The above example works pretty good. But there are some issues:

- As mentioned earlier, there are several strategies to perform Multi Label Classification. Lots of experimentation is required.
- Need to perform a Hyper Parameter Search to optimize performance.

- Requires a powerful GPU and lots of time to train.
- Would take additional time and effort (and skill) to move this model to production.

The above issues pose a great limitation to moving such models quickly into deployment. Luckily, there is an easy alternative.

Nanonets to the rescue!

Nanonets provides an easy to use API to train a Multi Label classifier. It takes care of all of the heavy lifting, including Data Augmentation, Transfer Learning and Hyper Parameter search on their GPU clusters. It does all of this within an hour, and provides a REST API to integrate the model with your services. They also provide an annotation service if required.

It is pretty easy to get started with the Nanonets API. This section gives an overview about the steps involved in setting up the API to perform the same Multi Label food classification experiment. For a more detailed set of instructions, check out the GitHub repository over here.

Part 1—Setup

Clone the GitHub repository. Obtain a free API key from Nanonets, set the appropriate environment variables, and run `create_model.py` as explained in the repository.

Note: In `create_model.py` we have to specify the list of possible labels (in our case, 10 food categories). I have already specified the labels in the code so you can directly

Part 2—Upload the Dataset

Nanonets requires the dataset to be provided in the following directory structure:

```
-multilabel_data
  |   -ImageSets
  |   |   -image1.jpg
  |   |   -image2.jpg
  |   -Annotations
  |   |   -image1.txt
  |   |   -image2.txt
```

I have already created the dataset in this format and provided a download link (and some instructions) in the GitHub repository. By running `upload_training.py` , the data is automatically pushed to Nanonets.

Part 3—Training and Inference

Once the dataset is uploaded, you can execute `train_model.py` to start the training process. The script `model_state.py` will keep you updated about the current state of the model. You can also checkout the status of your model from your user page at Nanonets as shown below

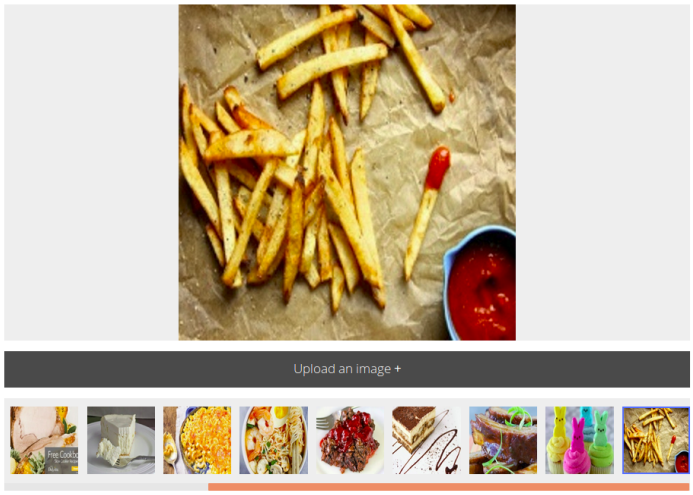
Nanonets

Images per category:	carbs: 1240 dessert: 418 fats: 1009 healthy: 355 junk: 411 mains: 797 meat: 382 protein: 709 soups: 118
Accuracy:	75.766014%
API calls this month:	554 / 1000
Plan	DEVELOPER
Pricing	$0 FOR 1000 API CALLS

Go to Model | Download Code to Use Model | Python

Status of your model in Nanonets

Once your model is trained, you can run `prediction.py` to use the deployed model! You can also observe the sample JSON output from your user page as shown below.



JSON RESPONSE


```
{
  "message": "Success",
  "result": [
    {
      "message": "",
      "prediction": [
        {
          "label": "carbs",
          "probability": 0.9797203
        },
        {
          "label": "junk",
          "probability": 0.89160675
        },
        {
          "label": "fats",
          "probability": 0.888279
        },
        {
          "label": "appetizer",
          "probability": 0.82973194
        },
        {
          "label": "protein",
          "probability": 0.017884376
        },
        {
```


Sample JSON response as shown by Nanonets.

Performance

epochs in minutes is plotted in the below bar graph.

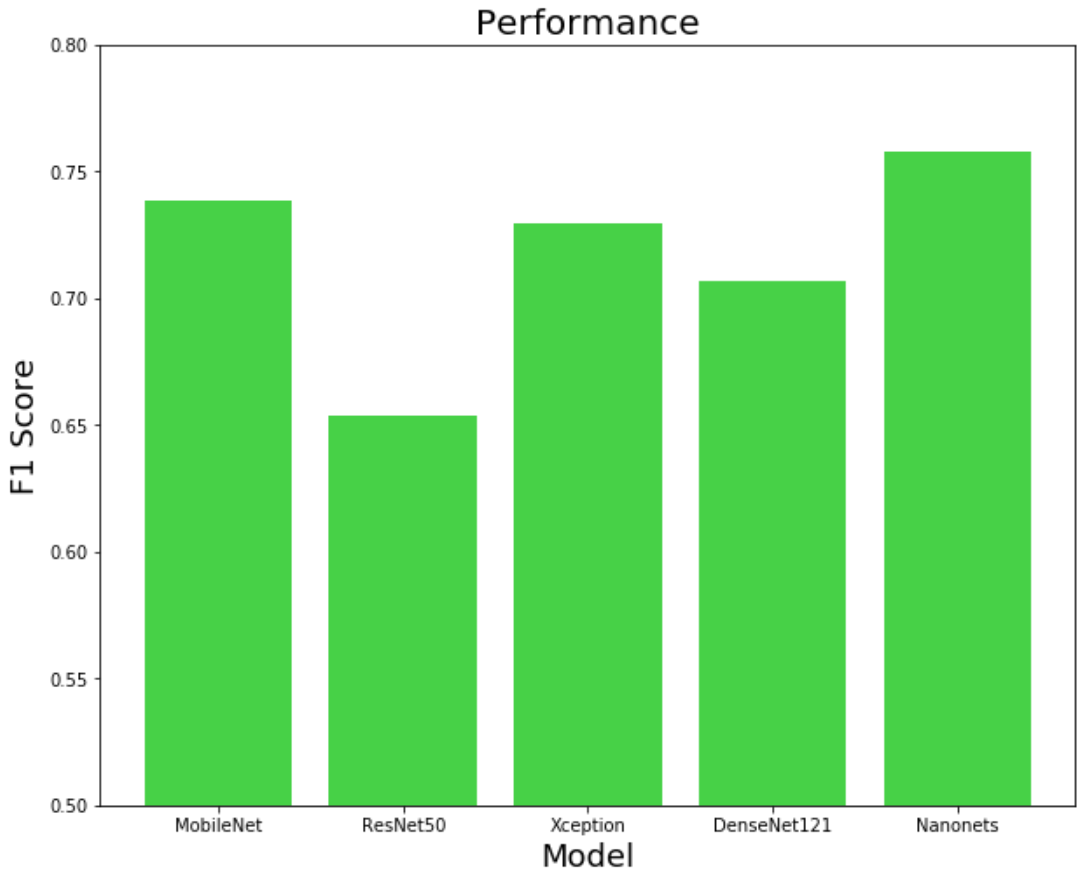
The MobileNet is the fastest to train owing to its efficient architecture. Unsurprisingly, the Xception network takes a lot of time as it is the most complex network among the ones we compared.

Training Time in Minutes.

Do note that the training time **does not** account for the time incurred for hyperparameter search, model tuning and model deployment. These factors greatly add on to the time required to move a model to production. However, Nanonets provided a production ready model within 30 minutes, even after accounting for all of the above factors.

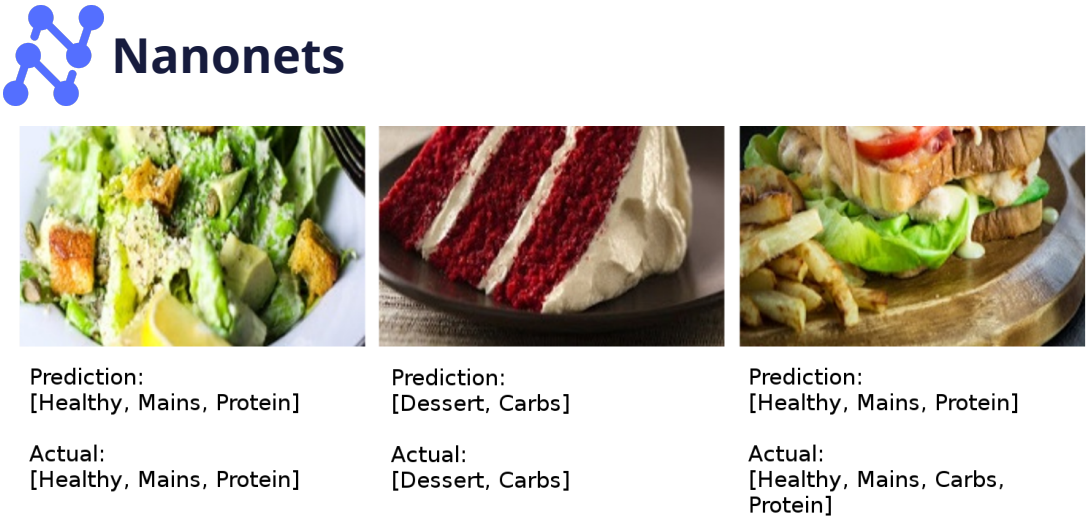
Nanonets

plot the F1 score obtained by the various Keras models and Nanonets.



F1 Score of various models.

Nanonets clearly has a higher score than the Keras models. Surprisingly, the MobileNet model came very close to catching up. Due to its parameter efficient architecture, it can mitigate overfitting better compared to the other Keras models. The relatively lower score of all models can either be attributed to the complexity and limited size of the dataset, or to noisy annotations. Let do a visual observation of the output as well.



Predicted vs Actual labels

Looks like our training was pretty successful! By using a larger dataset we could achieve better performance. You can also further experiment by using different datasets for other innovative applications by applying the concepts discussed above.

Lazy to code? don't want to spend on GPUs? Head over to Nanonets and start building your custom multi label classification models for free!

Learn more

Login

Add a comment

Nanonets

NEWER POST

How To Make Deep Learning Models That Don’t Suck

OLDER POST

Content Moderation in 2020 : Human vs AI

Nanonets

PRODUCTS

Object Detection

Image Classification

SOLUTIONS

NSFW

Drones

E-commerce

Inspection

Multi Label Classification

OCR API

Hygiene & Safety Compliance

Insurance

Nanonets

Solar Panel faults

Windmill faults

NSFW Content Moderation

Furniture Research & Recommendation

Blog

CONTACT

156 2nd Street, San Francisco, CA 94105, USA

+1 650 382 8676

info@nanonets.com