Capstone Project - CNN Classifier **Bas Donker**
Machine Learning Engineer Nanodegree **2018/02/08**

I. DEFINITION

Project Overview

More than half the population owns a smart-phone or other camera-enabled mobile device they carry with them for most of the day. Digital storage is cheaper than it has ever been and continues to drop in price per gigabyte. Taking pictures has never been easier. On top of that, social media incentivizes us to share those pictures with our friends. Needless to say, we take a lot of pictures now[1].

Just a few decades ago, cameras were expensive. The film on which the photos were taken also didn't come cheap and were limited in size. To be able to see the actual pictures, you would have to go to a shop to have them developed. Looking at the difference between then and now, it is easy to see why there has been a shift in the way we handle pictures. We used to carefully put our photos in photo albums in either a chronological or categorical order. Now, we just upload them to our favourite social media site and forget about them as soon as the stream of likes start to decline. And what about all the pictures that weren't put on the internet? Those seem to be doomed to live on your DCIM[2] folder on your phone's memory forever.

Problem Statement

Having your pictures neatly organized into different subdirectories has its advantages. It is much easier to find that impressive picture of your dog you took 2 years ago if you don't have to scroll through hundreds of selfies and pictures of food to get there. But with the huge amount of pictures an average smart-phone contains now, it takes too much time to organize them all.

In this project I created an application that can sort these pictures automatically, based on what the images contain. Pictures of landscapes are copied into the '*landscapes*' directory, pictures of groups of people go in the '*group photos*' directory and pictures that contain my face will end up in the '*me*' category.

The application uses an image classifier that was trained on a larger dataset. I'm using transfer learning[3] for the final categorisation of the images.

The purpose of this application is to organize photos in the way the user intends. Different users may want to organize their photos in different ways, so the application needs to be as flexible as possible. These tasks are involved in creating this application:

1. Create a root directory for all the original images: 'images/original/'.
2. Create subdirectories for all the categories the classifier needs to be able to recognize (Animals, Food, Landscapes, etc.)
3. Populate these folders with images of the specified categories.
4. If there aren't many images, find more training examples on the web until there are around 100 images in each category.
5. Make the application go through all the original images, and split them up into training, validation, and test sets[4].
6. The application will augment the images multiple times, rotating them randomly and / or flipping them horizontally.
7. Download some pre-trained models to use transfer learning on.
8. Train the classifier for each of the models.
9. Save and use the model that is able to classify images in the test set most accurately.
10. Run the classifier on all pictures in the camera roll to categorize the images

---

[1] https://mylio.com/true-stories/tech-today/how-many-digital-photos-will-be-taken-2017-repost
[2] https://en.wikipedia.org/wiki/Design_rule_for_Camera_File_system
[3] http://ruder.io/transfer-learning/
[4] https://machinelearningmastery.com/difference-test-validation-datasets/

My Dataset

I will be using a dataset I have created myself. The data consists of a combination of pictures I've taken as well as some images I found using Google Image Search, to make sure we have enough images to train on. I have categorized the images as follows:

- animal (116 images)
- city_scape (100 images)
- food (68 images)
- group (79 images)
- landscape (61 images)
- me (40 images)

Metrics

To determine how well a classifier is doing, we will calculate its F1 score[5]:

$$\mathrm{F1} = 2 * \frac{\mathrm{precision} * \mathrm{recall}}{\mathrm{precision} + \mathrm{recall}}$$

The higher the F1 score, the better the model is at classifying the images correctly.

And advantage of calculating the F1 score rather than the accuracy is that the F1 score is irrespective of how many images are in each category. For instance, if we have 1000 pictures of food and only 10 pictures of landscapes, a classifier that always predicts 'food' will be correct 99% of the time, even though it clearly hasn't learned how to properly classify the different images.

[5] https://sebastianraschka.com/faq/docs/multiclass-metric.html

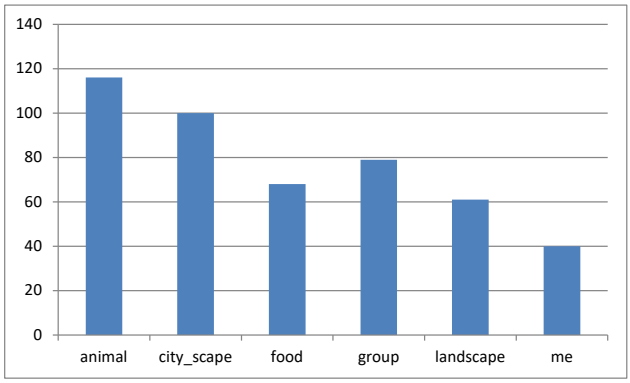
II. ANALYSIS

Data Exploration

I have created a dataset of a combination of pictures I have taken myself and pictures I found from other sources. All pictures are categorised in subdirectories that have the same name as the labels these pictures need to be given. For example, in my own dataset:

- images/ *root directory*
  - original/ *original dataset*
    - animal/ *pictures of animals*
      - dog.jpg
      - zebra.jpg
    - city_scape/ *pictures of city streets*
      - dublin.jpg
      - new_york.jpg
    - food/ *pictures of food*
      - burritos.jpg
      - that_thing_in_the_fridge.jpg
    - group/ *group pictures*
      - colleagues.jpg
      - me_and_friends.jpg
    - landscape/ *pictures of landscapes*
      - hills.jpg
      - mountains.jpg
    - me/ *pictures of me*
      - handsome.jpg
      - really_handsome.jpg

My dataset consists of *464* images. This is how these are distributed per category:



As you can see in the graph, the images are not distributed evenly. The difference between the category with the most images (animal) and the least images (me) is 76.

This is why it's important to measure the model's performance in terms of *F1 Score.* Otherwise, if the model just predicts 'animal' for each image, it will still be correct 25% of the time.

Some of the images need to be classified as multiple categories. For instance, a group picture with me in it needs to be categorized as both '*group picture*' and '*me*'. Such images are copied into the same subdirectory for training.

For example, this is not just a picture of me, but also a picture of a landscape, so this picture is both in the '*me*' and '*landscape*' directories.

All of these images have different dimensions and resolutions. They will all need to be scaled to the same resolution (256, 256) for the classifier to train on.

The images are in RGB colours, which means we have to introduce 3 additional dimensions; one for each colour channel, making the final matrix representation 256 x 256 x 3.

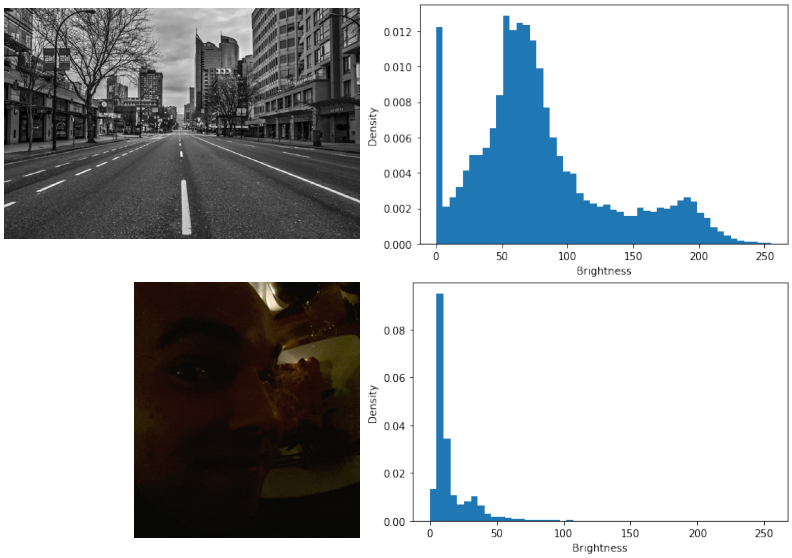
Because we have a relatively small dataset, I will augment[6] these images to generate more data to train on. Each image will be augmented 20 times, with random rotation and / or flipping them horizontally.

original *augmented_1* *augmented_2* *augmented_3* *augmented_4*

Visualization:

Since our images are taken with a variety of different devices, come from different sources, and are taken in different lighting situations, we can expect the pictures vary significantly from each other in terms of colour saturation and exposure.

Let's look at two examples:



[6] *https://machinelearningmastery.com/image-augmentation-deep-learning-keras/*

In the first example, there is a wide range of brightness values. All values from 0 to 255 are represented in the image. The second example is very dark. The brightness of that image doesn't go much beyond 100 (not even half the maximum brightness).

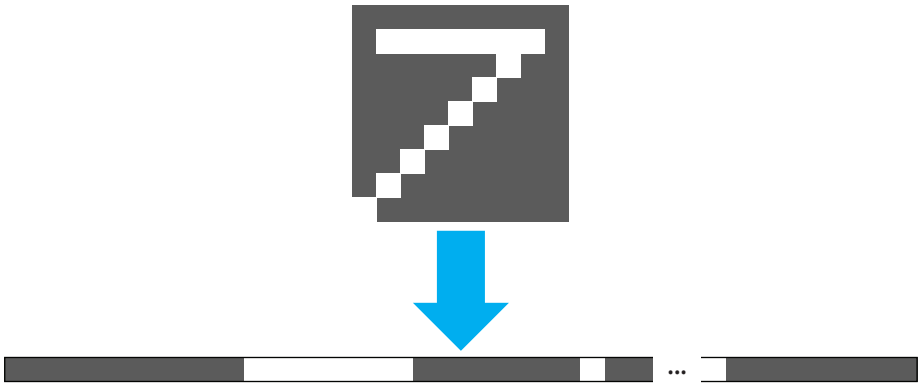
We need to keep these differences in mind when preprocessing our data if we want our model to generalize well. For instance, if most group photos are taken in dark places, the algorithm might start classifying all dark pictures as group photos.

Algorithms And Techniques

To classify images based on their contents, it's best to use a Convolutional Neural Network, or CNN[7], to train a classifier.

The pre-trained models are Convolutional Neural Networks. They are similar to traditional Neural Networks, but the difference is that Convolutional Neural Networks are able to retain spatial information, which is useful for images.

When training a traditional Neural Network on an image, it needs to 'flatten out' all the pixels in that image into a long array of numbers:



While the network is capable of learning using this format, this technique comes with a number of problems. For instance, if the picture is slightly translated, the input array looks a lot different.

Convolutional Neural Networks solve the problems detailed above.

These are the main components of a CNN:

1. The input layer
2. A set of *kernels* with *filters*
3. Pooling layers
4. Fully connected layers
5. The output layer

The input layer of the network takes an image in the form of a 3 dimensional numeric matrix. Two dimensions for the height and width of the image, plus 3 dimensions for each colour channel (R, G, and B). So if our image is 256 by 256 pixels, the input matrix is of size (256, 256, 3).

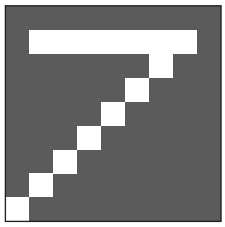
The kernels and filters are the meat and bones of a convolutional neural network. A kernel is a small 'window' that slides over the whole of the image. A kernel is made out of another numerical matrix, called a *filter*. Filters are used to detect specific patterns in images.

To illustrate, let's look at a simplified example of this process.

---

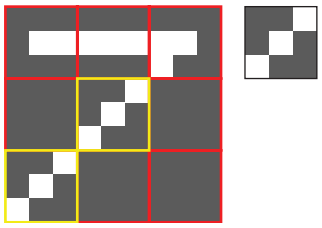
[7] http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks

Say that this is our input image:



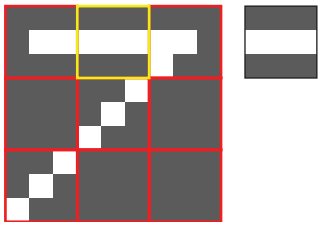
We will apply the following kernel filter to this image:

With this filter, we are looking to see if the input image contains any horizontal lines that slope upwards. We will slide this filter over the image and mark any sections that contain this filter as *yellow* and sections that do not contain what the filter is looking for as *red*.



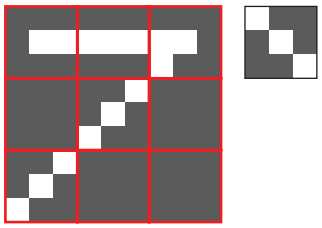
There are two sections that contain what the filter is looking for (the middle and bottom left).

Let's apply a different filter to see if the image contains what that is looking for:



Here we are looking for a horizontal line and we found that the image does contain a section like that (top centre).

Let's apply the first filter again, but this time flip it horizontally:



There are no sections in the image that contain a downwards sloping diagonal line, so we can say that this image doesn't contain that feature.

This is just a simple example. In reality, the filters and the patterns they look for are much more complex[8].

This process closely resembles how we as humans look at photos. When we look at a picture, we determine what is in the picture based on what features it has. We might see windows, a door and a pointy roof and determine

---

[8] http://cs231n.github.io/understanding-cnn/

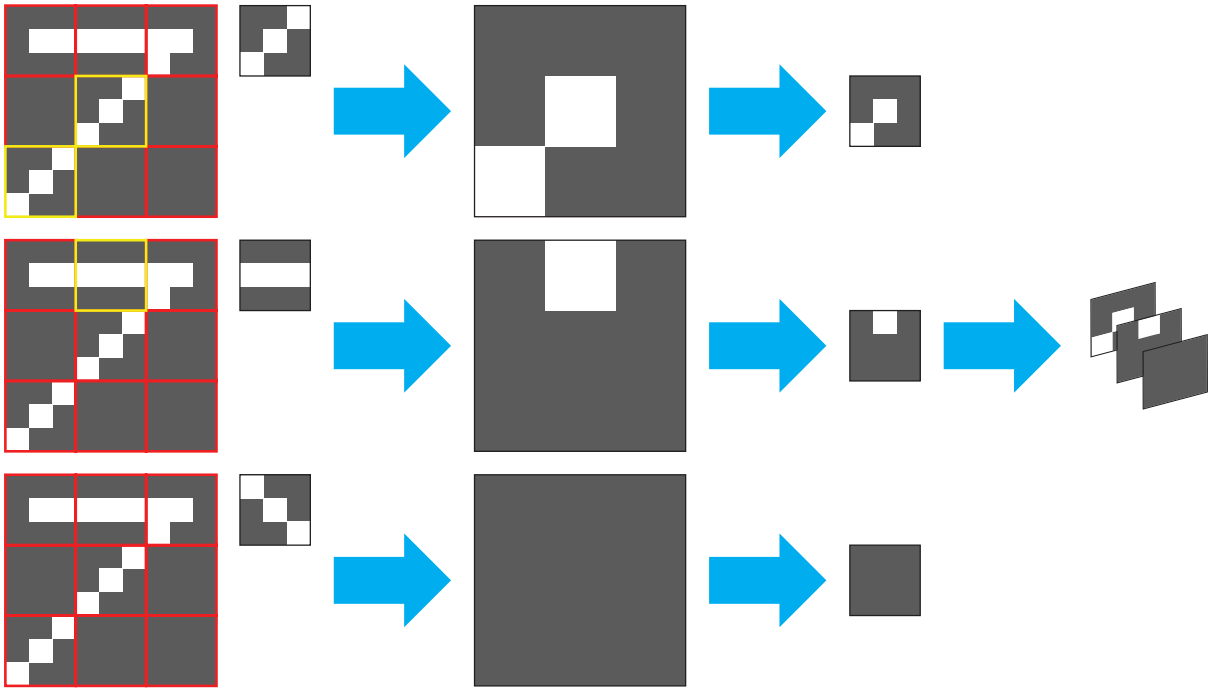
that this must be a picture of a house.

The patterns in the filters are not something that we need to create ourselves. During the training phase, the algorithm will create these patterns and will figure out what visual features are most important in the images.

The operation of taking a filter and sliding it over an image to see if any areas of the picture match the filter are done in the Convolutional Layers[9] of the CNN.

After sliding this kernel with filter over the image, we can create a smaller version of the original image by just noting where the filter found the pattern it was looking for, and where it didn't. Where a strong resemblance of the pattern was found, we colour the image bright and where little or no resemblance was found, we colour dark. Then, we take those re-sampled images of each filter and arrange them in the depth dimension of the output matrix.

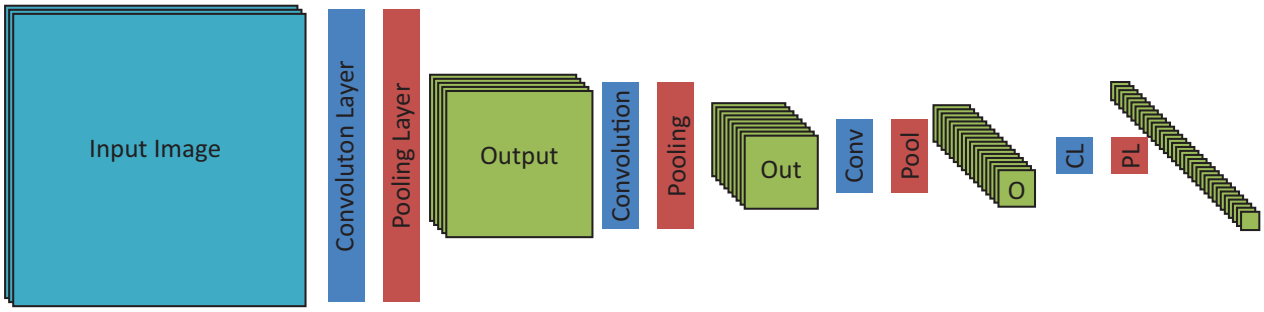
Back to our example image and filters:



The process of scaling down images this way to the core of its features is done in Pooling Layers[10].

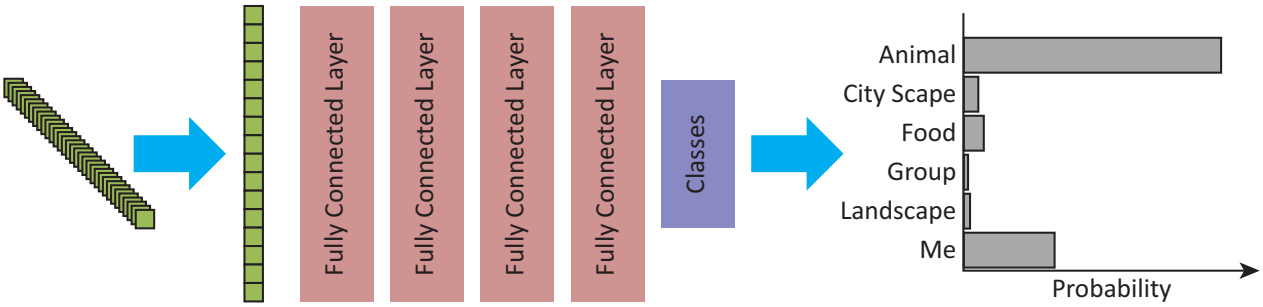
After each set of Convolutional and Pooling layer, the images get smaller, but the depth of the matrix becomes larger. After enough iterations, we will have images of a single pixel, but the matrix will be 65,536 (256x256) deep.

This is a visual representation of how the convolution and pooling layers affect the inputted image:



[9] http://cs231n.github.io/convolutional-networks/
[10] https://leonardoaraujosantos.gitbooks.io/artificial-inteligence/content/pooling_layer.html

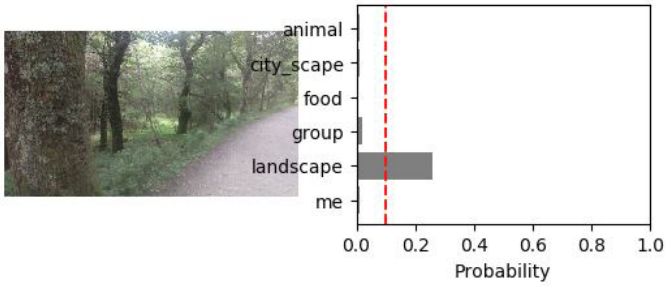
Once the image is as small as a single pixel, we can add a *Fully Connected* layer (or several) to the model to make the final predictions on what class the image is of.



Since we are using pre-trained models, we don't have to train the filters ourselves. We already have the filters that look for patterns in the images we pass to the algorithm.

The pre-trained models already have fully trained convolutional and pooling layers, so we only need to connect new fully connected layers and train its weights to make the final determination based on what visual features the convolutional layers found.

When we have our trained neural network, we can pass new images to it and have it calculate the probabilities of what category the image should be in.



Because we want to classify some images into multiple categories (a picture of me and a group of friends holding food in the middle of a city street will need to be classified as 'group photos', 'me' and 'cityscapes') and some images won't fit in any of the categories (a picture of a football will not be in any category), we can't simply classify the images based on what category is the most likely. Instead, we'll use a classification threshold. In the image above, I've set a threshold of 0.1 (10%). This means if the model is more than 10% confident the image belongs to a category, it will classify the image as that category.

If a picture is a group photo that includes me, it might have a 70% probability of it being a group photo and a 30% probability of the image containing my face and a 0.3% probability of the picture containing an animal. We can set the threshold to 10% and it will classify the picture as both being a group photo and a picture of me, but not as a picture of an animal.

If we pass it a picture of a football, the probabilities might turn out like this:

- Food: 0.04%
- Landscape: 1.50%
- Cityscape: 0.78%
- Me: 8.41%
- Group photo: 0.90%

In this case, none of the categories are over 10%, so the image is not classified in any of them.

After we trained our model, we can look at how confident it is when classifying the images in the testing set to determine what the threshold should be.

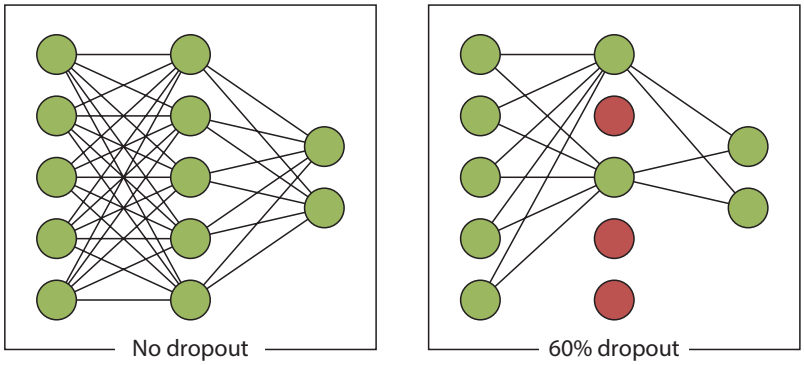
For the transfer learning model, we'll use these parameter values:

- Hidden layers: 1
- Hidden layer nodes: 256
- Dropout: 0.4
- Activation: Sigmoid

Dropout:
We want our model to be able to look at an image and predict what category the image should be in. We don't want our model to just memorize the images it has seen in the training phase.

When a model starts memorizing images rather than learning generic features, it starts to overfit. It will have excellent results on images in the training set, but it might not be able to categorize images it hasn't seen before.

One way to combat this, is by adding a *dropout layer*. With a dropout layer, we take a percentage of random nodes for each step and deactivate them. This way, the model is forced to 'forget' some of the information it learned.



Benchmark

To test the effectiveness of using transfer learning, I will also create a Convolutional Neural Network from scratch and compare its output to the transfer learning models. The convolutional neural network will have this structure:

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 256, 256, 16)      208
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 128, 128, 16)      0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 128, 128, 32)      2080
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 64, 64, 32)        0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 64, 64, 64)        8256
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 32, 32, 64)        0
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 32, 32, 128)       32896
_________________________________________________________________
max_pooling2d_4 (MaxPooling2 (None, 16, 16, 128)       0
_________________________________________________________________
dropout_2 (Dropout)          (None, 16, 16, 128)       0
_________________________________________________________________
flatten_1 (Flatten)          (None, 32768)             0
_________________________________________________________________
dense_5 (Dense)              (None, 512)               16777728
_________________________________________________________________
dropout_3 (Dropout)          (None, 512)               0
_________________________________________________________________
dense_6 (Dense)              (None, 512)               262656
_________________________________________________________________
dropout_4 (Dropout)          (None, 512)               0
_________________________________________________________________
dense_7 (Dense)              (None, 6)                 3078
=================================================================
Total params: 17,086,902
Trainable params: 17,086,902
Non-trainable params: 0
_________________________________________________________________
```

We will compare the F1 score of our own convolutional neural network to the F1 scores of the transfer learning models to see which performs better in terms of accuracy.

Complications

Some complications arose while developing this application, mainly to do with hardware resources.

Augmentation

The initial plan was to augment each image 25 times, and then store the augmented versions in the dataset, along with the originals. This meant that our 464 images now turned into 12,064 images (11,600 augmentations plus 464 originals). While it's good to have this many images and there is plenty of space available on my hard drive, this became a problem when trying to load all of them into memory for testing.

The solution to this was to make an augmentation generator that was separate from the original images. This way the algorithm can take an image from memory, augment it, and remove it from memory once it's done with it.

Low Image Quality

After having solved the memory problems of loading all augmented images into it at once, I trained the model for a few iterations and I discovered that it wasn't performing well at all. It was barely performing better than random guesses.

I turned off the augmentation generator I built and trained the model again. This time, it was doing much better, but stopped improving before achieving acceptable results, due to the dataset being too small.

To get a better understanding of what the augmentation generator was doing, I decided to augment some images with it and this time save them as image files. Looking at the augmented versions, I saw the quality of these images were very poor. There were a lot of jagged edges and everything was very pixelated. I myself had a hard time distinguishing what the images were of.

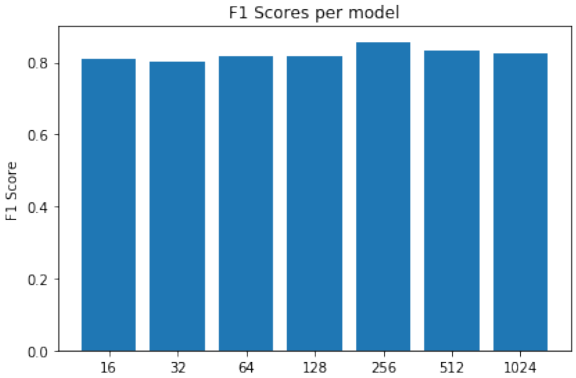
The cause of this problem was the fact that I first preprocessed the original images, scaling them down to 256 by 256 pixels, and then augmented them after. I needed to do this in the reverse order to keep the image quality (first augment, then scale down to the target size).

Too Many Weights

Originally, I wanted to add multiple fully connected layers to the model, each with around 500 nodes. Because of this, the amount of weights the model had to manipulate during training rose to nearly 70 mln. It was very apparent that I do not have the hardware resources available in my own PC to perform calculations of that scale.

One way to solve this problem is to use AWS, so that the bulk of these calculations are done with more capable hardware, but I want to keep this application usable for people who are less tech-savvy.

Since AWS wasn't an option, I did some tests to see if adding more nodes would actually improve the model. I implemented different configurations and trained the model for a few epochs. These were the results:



As it turned out, even having just one additional hidden layer with only 16 nodes did pretty well. I choose a

middle ground and went with 256 nodes (even though the difference in F1 score is negligible).

Probability Sum = 1
After the issues described above were resolved, the model worked very well. Training was relatively quick and it achieved F1 scores of over 0.85 on the testing set. It was time to run the model on a big collection of images I have, so I could see how well it did.

I then discovered there was another problem with the model. If an image doesn't fit either of the categories the model is trained on, it will still output a probability matrix of which the sum is equal to one. For instance, if I trained the model on categories 'dogs' and 'cats', it will do very well on images of dogs and cats. But if I then show it a picture of a bee, it might output a probability matrix like this:

- dog: 0.44
- cat: 0.56

The reason for this is that I was using softmax[11] as my activation function in the final layer. Softmax computes the probability of each outcome in a way that its sum is always equal to *1.0*.

Softmax is asking the question "*Which category out of 'cat' and 'dog' is most likely correct?*".

What we need is an activation that asks for each category "*How confident am I that this image contains a cat? How confident am I that this image contains a dog?*".

After changing the activation function from softmax to *sigmoid*, the model stopped classifying images of footballs as my face.

How many Epochs?
A small, but not irrelevant issue was to figure out how many epochs the model needs to train in order to achieve a good result. For my particular dataset, about 20 epochs seemed to be enough, but I wanted the application to be flexible enough to fit other datasets. I could keep the number of epochs very high, so we can be sure it works with any dataset, but that seems wasteful.

I resolved this by implementing the EarlyStopping[12] callback function in Keras. This essentially puts a cap on the amount of epochs the model will train. After each epoch, it checks to see if the validation loss has improved. If this value hasn't improved for a set number of epochs in a row, the model will stop training.

I set the value for the number of epochs very high (500) and the patience of the callback value at 20. This means that if the validation loss hasn't improved for 20 epochs in a row, the model will finish training.

---

[11] http://ufldl.stanford.edu/tutorial/supervised/SoftmaxRegression/
[12] https://keras.io/callbacks/

III. METHODOLOGY

Data Preprocessing

The steps to prepare the data for the neural network to train on are detailed in the "Image Preprocessor" jupyter notebook. The following steps are taken:

1. All the images in the 'originals' directory are put into an array
2. They are split up in training, validation and test sets
3. The images are converted into matrices of size 256, 256, 3 (image resolution is 256x256 and they are in RGB colour)
4. The pixel values are normalized so all values are between 0 and 1.

Implementation

For our transfer learning models, we will need to download the weights and structures of pre-trained models. This can be done directly from Keras. To train these models on our own categories, we need to strip off the top layers of these models and substitute our own. We will then load the preprocessed training and validation data into memory and train our models on that data. These are the steps used to train these models.

1. Using the Keras library, download the weights and structures of the following pre-trained models:
   ResNet50
   VGG19
   InceptionV3
   NOTE: When downloading these models, we can set the 'include_top' parameter to false. This means that the final fully connected layers will not be included so we can add our own.
2. We'll instantiate the three models and store them in a list called 'base_models'.
3. Load our preprocessed training and validation data into memory.
4. For each model in base_models, we'll add our own fully connected layer of the same size as the amount of categories we have.
5. Define the loss function, accuracy metric and callback method. The callback method should save the current weights if the loss function on the validation set improved.
6. Train the network on a small number of epochs.
7. After the last epoch, load the weights that has the lowest loss on the validation set and store them in a list called 'models'.
8. After all models are trained, calculate their respective F1 scores on the test set.
9. Save the weights and structure of the model with the highest F1 score, so it can be used later.

Now that we have chosen the best transfer learning model, we will create our own convolutional neural network and see how it compares.

1. Create a new Sequential model called 'ccn_new'
2. Define the model's architecture. We'll use 4 convolutional layers pared with maxpooling layers. After each maxpooling layer, the height and width of the image is reduced by 0.5 (256 > 128 > 64 > 32 > 16).
3. The final layer will be a fully connected layer, which will determine the final class of the images. This layer needs to be the same size as the number of classes we have.
4. Train the network on a small number of epochs.
5. After the last epoch, load the weights that had the lowest loss on the validation set.
6. Calculate the model's F1 score on the test set.
7. Compare this F1 score to the F1 scores of the pre-trained model and pick the model that scores the highest.

Refinement

After the training phases of all the models, we should know which one performs the best by looking at which one has the highest F1 score.

Using the model that performed best so far, we will tweak the following parameters to see what combination works best:

- Number of hidden layers
- Size of the hidden layers
- Learning rate
- Epochs
- Dropout values

For each combination of hyper-parameters, we will train the model for 1 epoch. If the F1 score of the model improved, we will use the combination of these parameters that achieve the best result.

IV. RESULTS

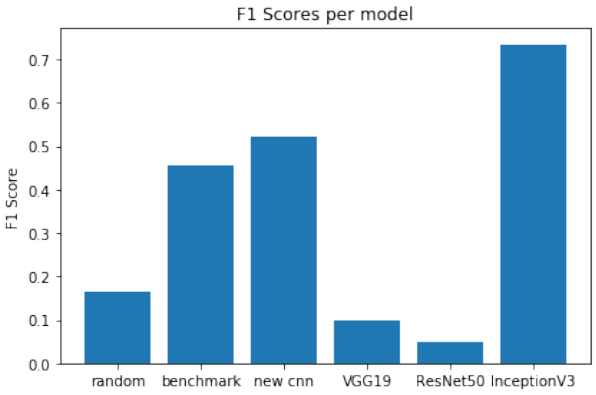
Now that we have a good understanding of what the model should do and how it should do it, it's time to train some models.

Choosing A Base Model

As outlined earlier, we have a number of different models to choose from that could potentially work very well. First, we need to figure out which of the following models we want to use:

- A random guesser
- A traditional Feed-Forward Neural Network (benchmark)
- A Convolutional Neural Network trained from scratch
- A pre-trained Convolutional Neural Network:
  - VGG19
  - ResNet50
  - InceptionV3

Apart from the random guesser, which obviously can't be trained, I trained all these options on a few epochs and compared their results:

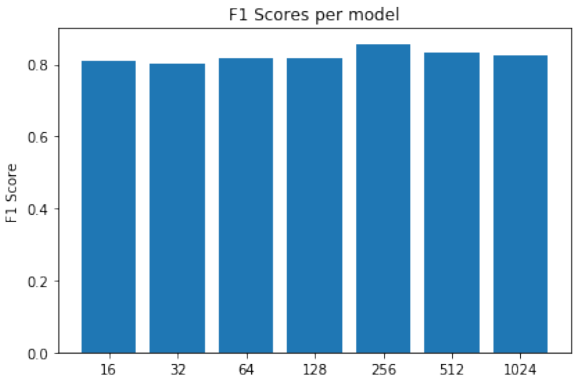


The pre-trained *InceptionV3* model is the obvious winner of this test, achieving an F1 score of over 0.7.

Optimizing the Chosen Model

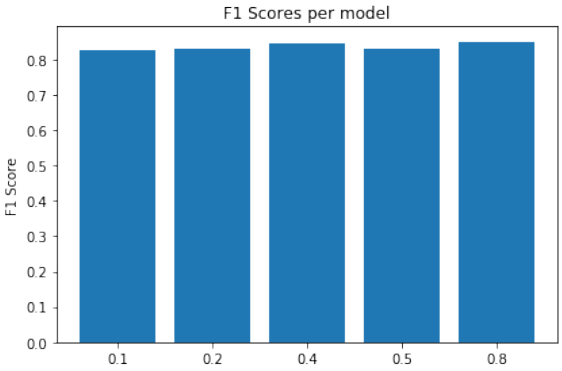
Now that we have determined that InceptionV3 is our best option, we'll see what we can do to make this model perform even better.

I first tried different sizes of the final hidden layer:



The differences in performance is very slight, so I choose a middle ground: 256 nodes.

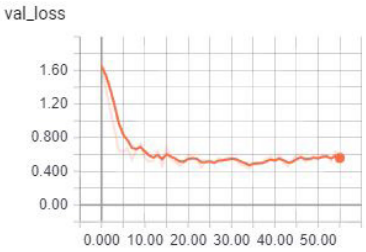
Next, I tried different values for the dropout layer:



Again, the results seem very similar, so I choose to use a value of 0.4, as it seems to perform well.

Training the Final Model

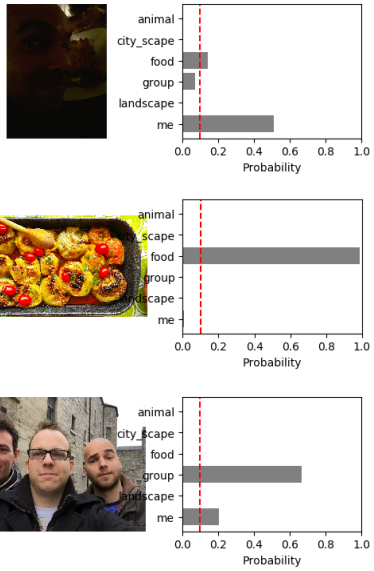
Now that the final model has been chosen, along with its optimal parameters, I trained the final model for a longer period of time (55 epochs) and monitored its validation loss in TensorBoard:



The validation loss dropped quickly in the beginning, but then levels out. The model achieved the best result around epoch 35. The final F1 Score of this model was *0.84*, which is a very good result.

Testing the Final Model

To get a better feel on how the final model is performing, I ran some test images (that the model hasn't seen during training) and plotted the model's predictions.

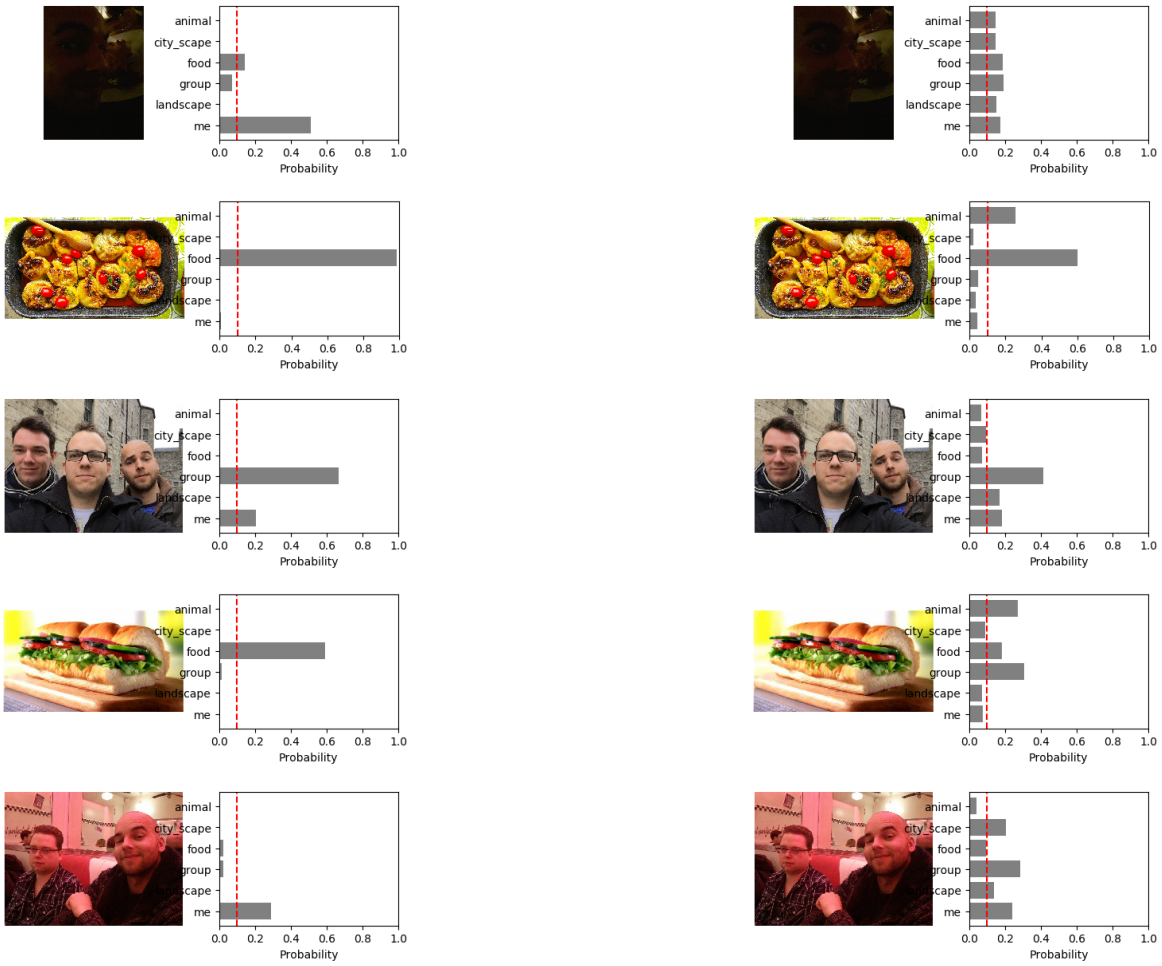


With a threshold of 0.1, it seems that the model is doing very well. Looking at the graphs, we see that it classifies most images correctly. There's one very dark image of my face with some food in the background. The model correctly classified the images as both a picture of me and a picture of food.

Justification

Lets compare the outputs of our final model to the ones of our benchmark model (the traditional Neural Network):

Inception V3 Benchmark



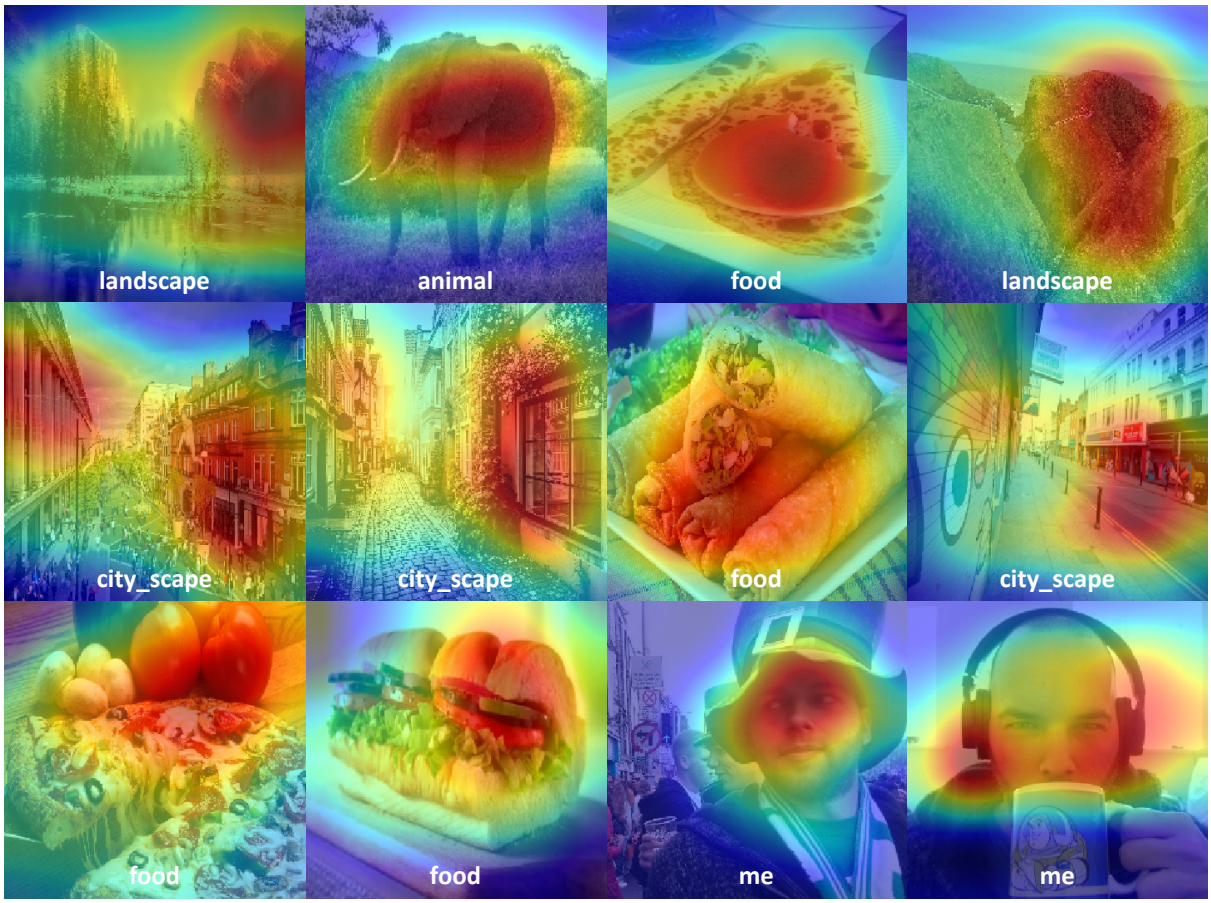
It's worth noting that if we wanted to classify these images using only the most likely class, the benchmark model isn't a bad option as it classifies a good amount of them correctly. However, since we want the model to be able to classify on multiple classes (or no classes) per image, we can see that there is no place where we can place the threshold that would achieve our desired result.

For instance, the benchmark model would predict the sandwich is an animal. Although the contents of that sandwich may have been an animal at some point, we can't give the model credit for that.

V. CONCLUSION

Free-Form Visualization

One neat thing you can do after training the model is to visualize the class activation maps. I have taken a set of images and visualized the class activation maps according to what the image classified as. The areas of which the algorithm was most convinced the image contains a particular category are highlighted in red.



Reflection

At its core, computer vision is actually pretty straight-forward and easier to understand than many people think. You take an image, run some filters on it to determine what features are in the image and finally decide on what the image is of based on all the features that were found.

It is interesting how closely the algorithm mimics how we as humans look at images. Seeing a lot of straight, parallel lines? That's a building! Is that fur, a long snout and floppy ears? That's a dog!

The difficult part of this project was finding a way to train a model with good results on limited hardware. Transfer Learning was a great help with this, as we don't need to train all the convolutional layers themselves, which is the bulk of the training phase.

What also added to the complexity was creating an algorithm that is able to classify an image into multiple categories (or no categories). Choosing sigmoid as the final activation function instead of the more commonly used softmax solved that problem (although it took longer for me to realize this than I'm willing to admit).

Improvement

Although I'm very happy with the actual results of the final model, some improvements can be made in the way the application is used. I tried to design a software package that is easy enough to use for people who might not know a lot about machine learning. Anyone who is able to install python and python modules should be able to train their own model and run it through their own pictures.

It would be nicer to package this application with a GUI, so even non-tech-savvy people are able to use it.

Another improvement that can be made is the classification threshold we set manually. I think it's possible to have the threshold set algorithmically using another machine learning process.

The amount of augmentations that are needed is reliant on how many images are in each category. If I have 1000 pictures of food and only 50 pictures of landscapes, it doesn't make sense to augment the food pictures the same amount of times as the landscape pictures.