Capstone Project - CNN Classifier **Bas Donker**
Machine Learning Engineer Nanodegree **2018/02/01**

I. DEFINITION

Project Overview

More than half the population owns a smartphone 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 incentivises 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 developped. 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 smartphone 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/

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:

- 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

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 colors, which means we have to introduce 3 additional dimensions; one for each color 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 signigicantly from each other in terms of color saturation and exposure.

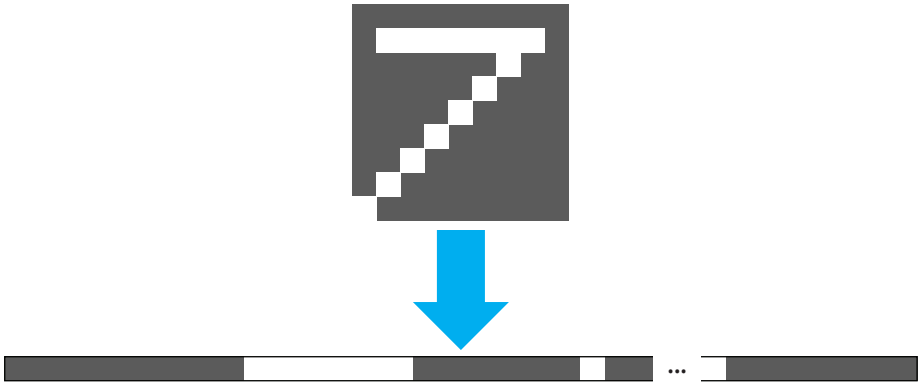
We need to keep these differences in mind when preprocessing our data if we want our model to generalize well. For instance, if most group photos are taken in dark places, the algorithm might start classifying all dark pictures as group photos.

Algorigthms 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.

---

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

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

These are the main components of a CNN:

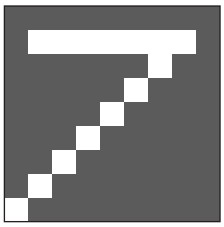
1. The input layer
2. A set of *kernels* with *filters*
3. Pooling layers
4. Fully connected layers
5. The output layer

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.

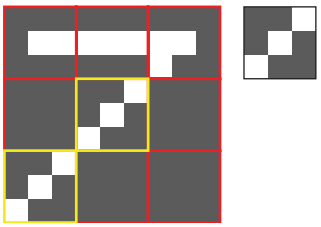
To illustrate, let's look at a simplified example of this process.

Say that this is our input image:



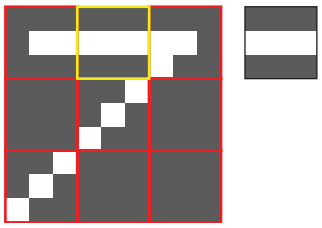
We will apply the following kernel filter to this image:

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*.



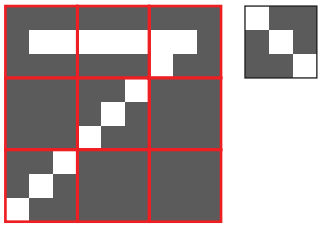
There are two sections that contain what the filter is looking for (the middle and bottom left).

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 center).

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



Ther 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 deterimine 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.

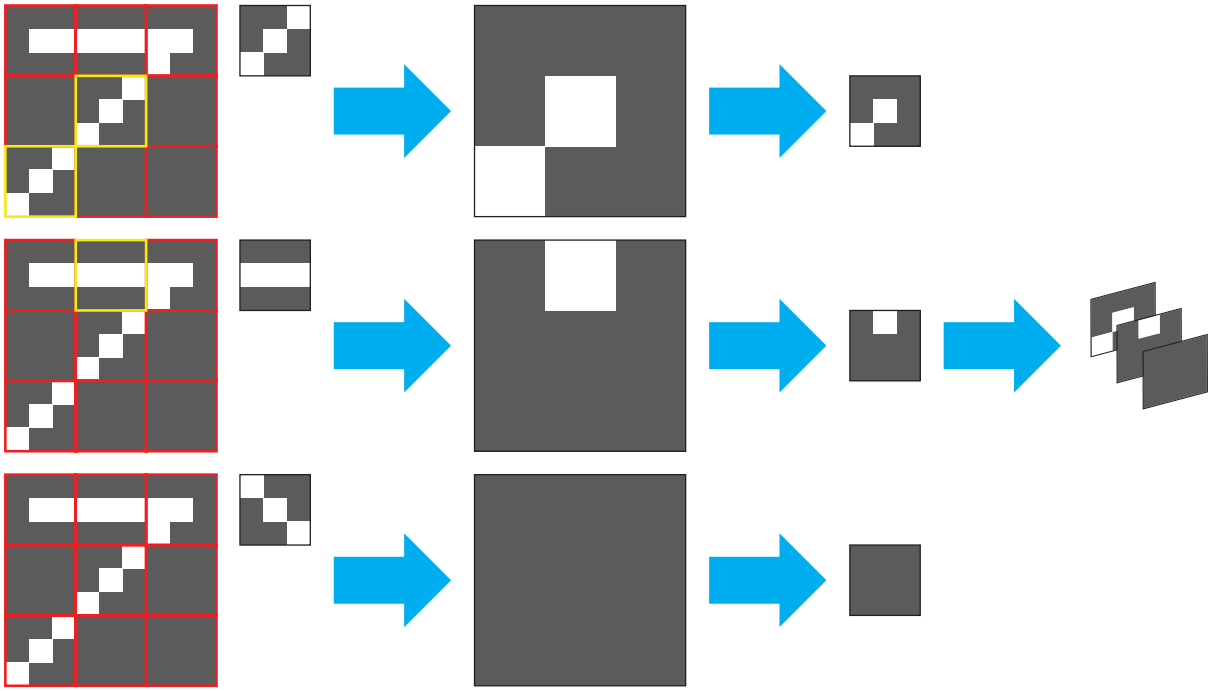
The operation of taking a filter and sliding it over an image to see if any areas of the picture match the filter are done in the Convolutional Layers[9] of the CNN.

---

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

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 resemblence of the pattern was found, we colour the image bright and where little or no resemblence was found, we colour dark. Then, we take those resampled images of each filter and arrange them in the depth dimension of the output matrix.

Back to our example image and filters:

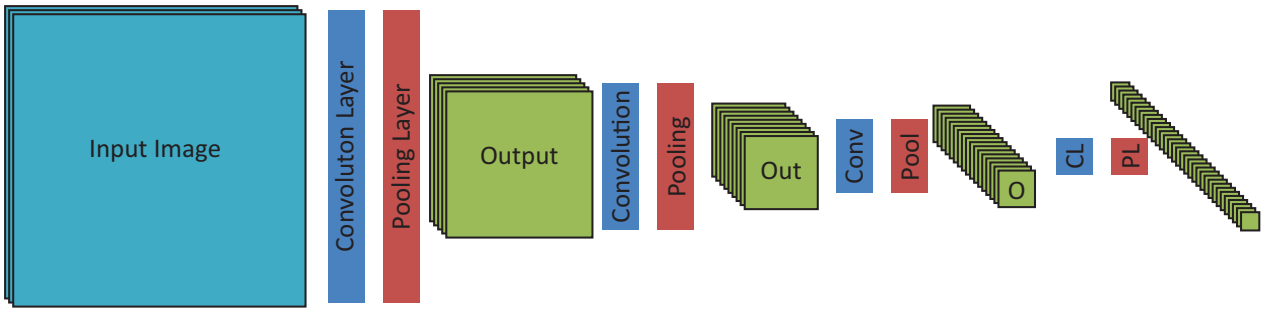


The process of scaling down images this way to the core of its features is done in Pooling Layers[10].

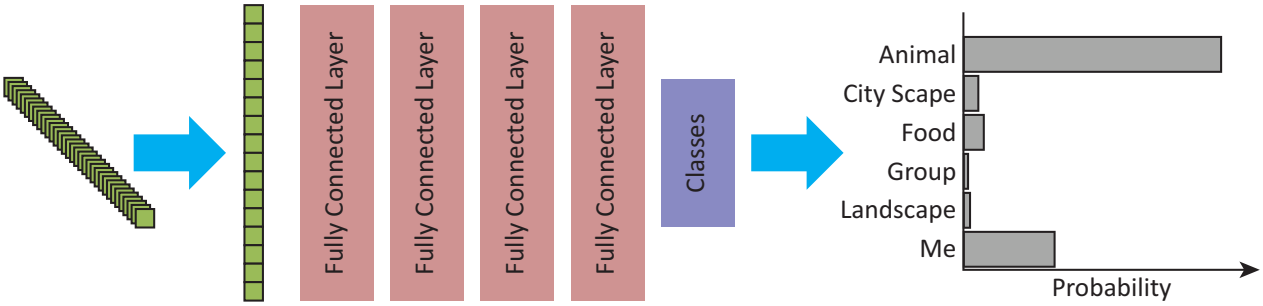
After each set of Convolutional and Pooling layer, the images get smaller, but the depth of the matrix becomes larger. After enough iterations, we will have images of a single pixel, but the matrix will be 65,536 (256x256) deep.

[10] https://leonardoaraujosantos.gitbooks.io/artificial-inteligence/content/pooling_layer.html

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



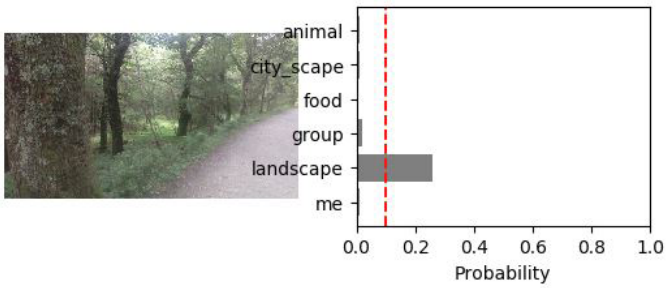
Once the image is as small as a single pixel, we can add a *Fully Connected* layer (or several) to the model to make the final predictions on what class the image is of.



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.

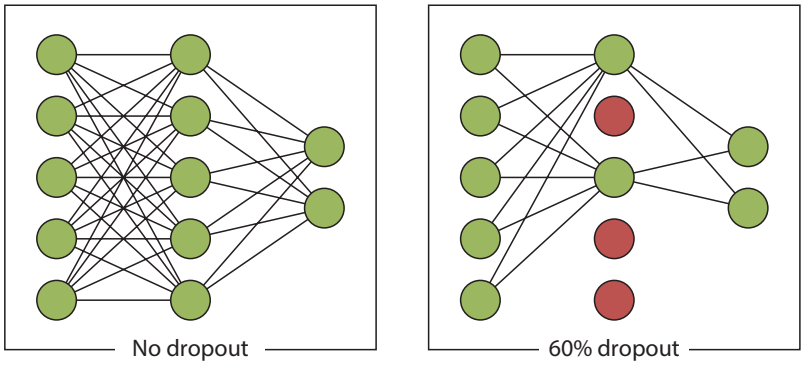
For the transfer learning model, we'll use these parameter values:

- 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.

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 color)
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 weigths 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 los 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

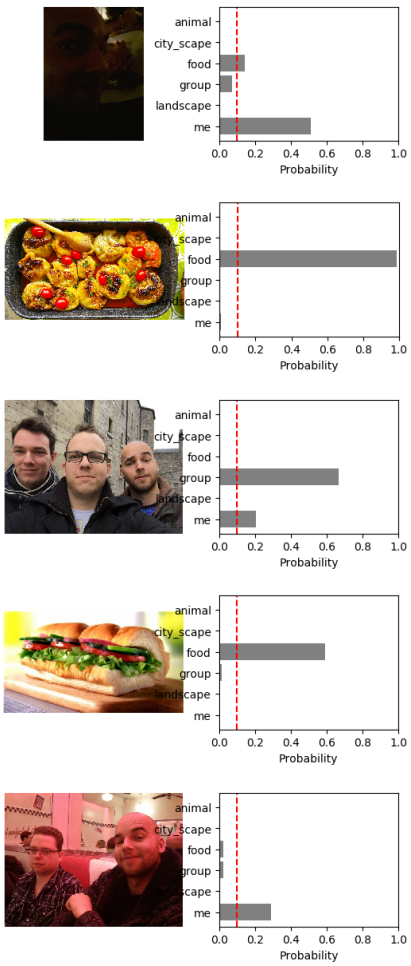
After the training phases of all the models, we should know which one performs the best by looking at which one has the highest F1 score.

Using the model that performed best so far, we will tweek the following hyperparameters to see what combination works best.

For each combination of hyperparameters, we will train the model for 1 epoch. If the F1 score of the model improved, we will store the values of the hyperparameters.

RESULTS

To get a good 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.