MACHINE LEARNING ENGINEER NANODEGREE

José Piñero
August 12, 2017

Capstone Project

I. Definition

This project is going to be enclosed in the Computer Vision domain. Computer Vision is an interdisciplinary field that seeks to automate tasks that the humans visual system do, vision tasks such as gaining full understanding of a scene or environment or recognizing objects from a picture. In general the computer vision tasks are enclosed in gaining or obtaining useful information from a given picture.

In order to accomplish such visual tasks, feature extraction and processing is needed in order to ultimately gain high-dimensional understanding from the presented data.

Precisely for this project I will be working on face recognition, a task where humans tend to have 97% or more precision, but it hasn't been that easy for machines until recently. This problem has attracted a lot of interest because of its applications in social media, mobile apps, security, among others.

A lot of work have been put into this task through the years and many algorithms have been developed to address it, from classical neural networks [1], genetic algorithms [2], principal component analysis [3] to name some. Recently a new approach named Convolutional Neural Networks has arisen, this algorithm have shown exceptional precision in vision tasks outperforming the traditional techniques.

There are a lot of datasets to chose from when dealing with face recognition problem, here are some of them:

- **The labeled faces in the wild dataset (LFW)**: This is one of the most famous datasets used in the field, it contains more than 13000 images from 1680 celebrities. All the pictures are taken from the web. The pictures are almost unconstrained, this means that you can see not only the face but also body and background. The only constraint is that all of the faces contained in the pictures were recognized using the Viola-Jones algorithm [5]. This dataset has a variant in which all the pictures are cropped.

- **SCface**: Is a database of static images of human faces. Images were taken in uncontrolled indoor environment using five video surveillance cameras of various qualities. Database contains 4160 static images (in visible and infrared spectrum) of 130 subjects. Images from different quality cameras mimic the real-world conditions and enable robust face recognition algorithms testing, emphasizing different law enforcement and surveillance use case scenarios.

- **CelebA**: This is a large-scale dataset, containing more than 200000 pictures from around 10000 celebrities. This is a very large dataset, and also contains 40 binary features annotators from the pictures, features such as (wavy hair, wearing hat, smiling, etc)

The number of existing datasets is big, that means that data is not a problem for this task. However, I've decided to use a custom dataset created some time ago by me and some friend. The dataset to be used has around 8000 images from 80 celebrities, all gathered from Google and then recognized and cropped by the usage of the Viola-Jones algorithm. It will be interesting to find algorithms capable of recognizing images picked by me, and see how well they perform. The dataset is located in the project repository, along with all the necessary files.

Problem Statement

The problem that I'm going to address in this project is the face recognition by using Convolutional Neural Networks (CNN). Given a picture of a face, correctly identify the person in it. This is a very challenging task due to the similarity between some persons. The general features are the same: nose, eyes, ears, mouth, etc, and that's why the algorithm must learn even more detailed features in order to correctly classify the person. I'm going to pass a set of pictures containing faces through the algorithm, in order for it to learn them and then try to correctly classify the known persons in any other pictures of their faces.

The general steps to solve the problem will be:

- First I'm going to build a random classifier in order to get a basic metric to start from. This classifier is going to try to recognize a person in a picture by random guessing. This algorithm isn't expected to achieve great performance and it's intended to show the behavior of a classifier that haven't learn anything.

- Secondly I'll build a basic CNN with a good general structure to analyze pictures, in order to show that this kind of algorithms are well suited for the task. With the usage of this CNN I expect to outperform the random classifier.

- Finally I'll use CNN at its maximum capacity by using one of the most popular techniques in the field, namely Transfer learning, in order to achieve the best possible results.

Metrics

This is a multi-class classification problem, and thus the performance of the algorithm can be measured using well known metrics such as precision or cross entropy, defined below:

$$Cross - Entropy = - \sum_{i=1}^{n} \sum_{j=1}^{m} y_{i,j} * \lg(p_{i,j})$$

Where:
- *n* stands for the number of instances in the dataset
- *m* stands for the number of different identities in the dataset
- $y_{i,j}$ is the true probability of the instance *i* to have the identity *j*
- $p_{i,j}$ is the predicted probability of the instance *i* to have the identity *j*

II. Analysis

Data Exploration

As mentioned before, the dataset consist of around 8000 pictures containing faces from celebrities. For each celebrity there are around 100 pictures, the faces may be in different angles, have different hair styles, skin tones and may wear hat or glasses. Here's an example:

Figure 1

The dataset have been created manually by searching public pictures from 80 celebrities and then automatically cropping them by using the Viola-Jones algorithm to keep only the face. Using this method for recognizing and cropping pictures adds some conditions to the dataset, for instance, all of the images are well lighted and they aren't blurry.

The 100 obtained pictures from each celebrity is going to be split into training, validation and testing set. In the following graph we can see the number of pictures used in each stage of the algorithms.

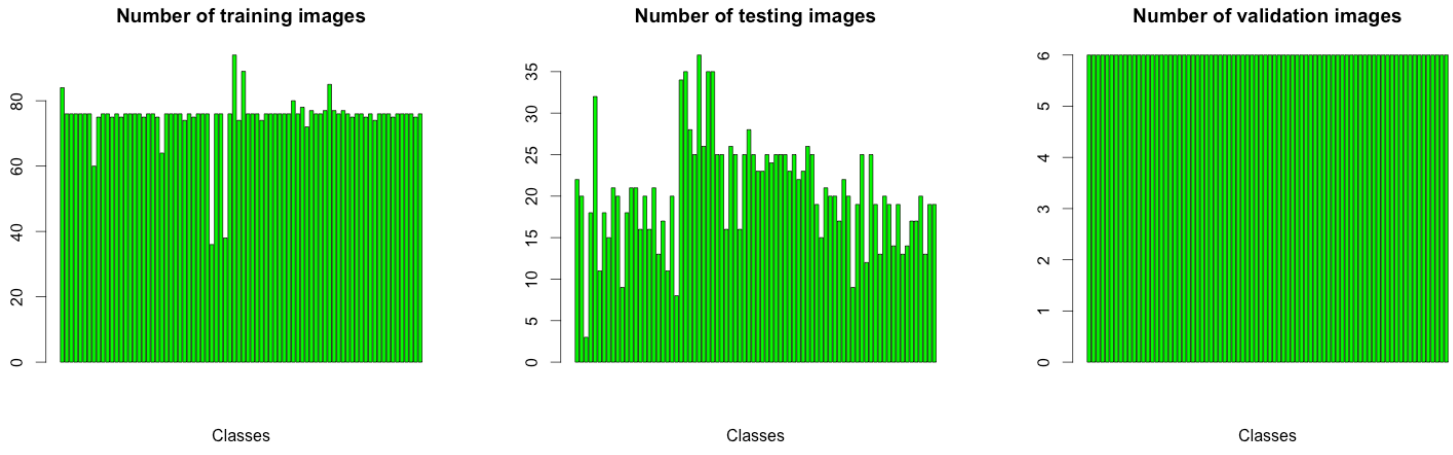


Figure 2

It can be noticed in the *Figure 2* that the number of training examples for each class is around 80, and the number of testing images is around 20, finally 6 pictures are taken from each class to test the resulting algorithms and get the accuracy for each model.

Algorithms and Techniques

The algorithms to be used are basically 2, a random model and a couple of different CNN's to compare results. Additionally a technique named Transfer learning is going to be used, in order to try to achieve state-of-the-art results on the dataset.

- **Random Classifier**: This algorithm is really simple since the only thing needed is to pick randomly a number between [0, 80) each time an image is passed. Here the number 80 stands for the number of different classes available in the dataset. There's not much more to explain here.

- **Convolutional Neural Networks:** The convolutional neural networks are very similar to ordinary traditional neural networks, the main difference resides in the usage of two new kind of layers together with the traditional fully connected layers. Heres how a traditional neural networks looks:

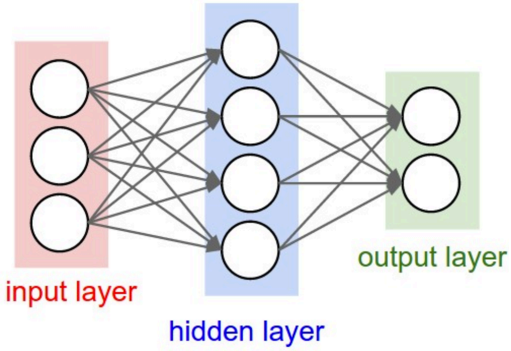


Figure 3

It can be noticed that the input layer is a 1-dimensional vector, and this doesn't help very much to learn spatial information that's usually contained in an image, and that's why a new kind of hidden layers were created, a layer capable of learning simple patters related to pixels relationships in the picture, these layers can be seen as filters. The interesting part comes when multiple number of these *special hidden layers* are stacked, since this allows the neural network to learn at each layer even more complicated patterns and finally learn not so trivial features from the image that can be used later to determine relevant information. These special layers are called *Convolutional Layers* giving its name to this kind of neural networks. Here's an image that illustrates the composition and behavior of this kind of layer:

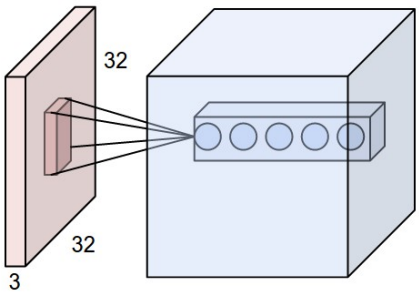


Figure 4

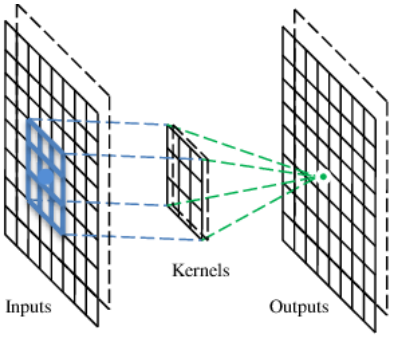


Figure 5

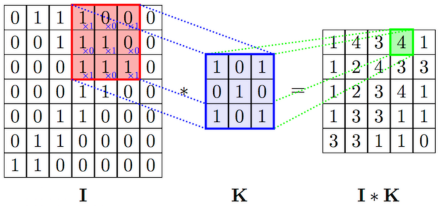


Figure 6

In the *Figure 5* we can se a filter (also called kernel) from the convolutional layer acting on an input data and producing an output data with the same dimensions as the input. The filter sweeps over the matrix horizontally and vertically making convolutions (shown in *Figure 6*) and building a new matrix with the results of passing the filter through the whole input. Finally, as there are more than one filter in each convolutional layer (each one creating its own output matrix) these output matrixes are stacked together to get a 3-dimensional matrix (shown in *Figure 4,* where each blue circle represents a filter and the blue cube is the output resulting from applying the mentioned filters to the input).

These convolutional layers are usually applied alongside with another special kind of hidden layers named *Pooling Layers.* This kind of layers are simpler than the previous ones, since they only apply a simple operation like *maximum, minimum, average, etc.* The purpose of this kind of layers is to reduce the dimensions of the data by subsampling it. Here's an image that illustrates the result of applying a pooling layer with the *maximum* operation.

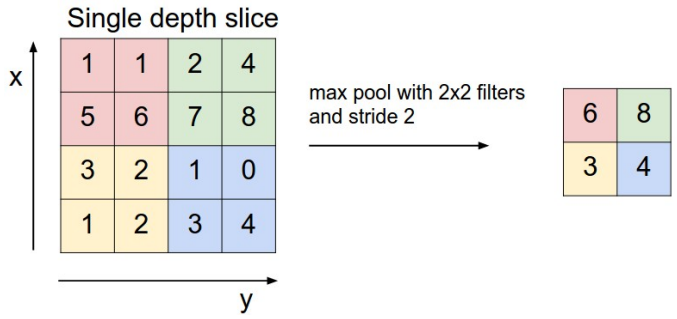


Figure 7

After applying the *Convolutional and Pooling layers* to the input data, a set of features is obtained typically in the form of a 1-dimensional vector, which can then be passed to a traditional neural network or another classifier to learn the weights for each l*earned convolutional feature* and finally classify the image.

This kind of networks may have several layers and become really deep in order to obtain better results, also they perform at their best when trained on large datasets. The natural consequence of this scenarios is that a lot of computational power is needed for training them and thus getting interesting results. The best known networks were trained during days on clusters with many GPU powered computers to parallelize computations. It is clear that training such networks is not possible for everybody and this is when the *Transfer Learning* technique comes to help.

The convolutional networks were found to generalize extremely well on the applied datasets. Allowing the network to extract the convolutional features from similar datasets with a lot less effort than training the whole network from scratch, this is called *Transfer Learning*. The technique consists basically on using the learned weights from a pre-trained convolutional network on a similar dataset without changing them (sometimes doing a light retrain) and changing only the last part of the network (the one in charge of classification) to represent the new problem classes.

In the project I'll use Transfer Learning to take advantage of two of the state-of-the-art convolutional networks in the field of facial recognition: FaceNet [6] and VGG-Face [7].

Benchmark

Currently I can't compare my results to anything since my dataset is custom, however some performance results can be mentioned to understand what can be expected.

- **Random results:** The random model has to guess from 80 classes each time an images is presented to it. It's clear that the expected accuracy to be obtained by this model is:

$$accuracy_{random} = \frac{1}{80} = 0.0125$$

Thus we can expect to get around 1.25% accuracy on this algorithm, not very useful.

- **Basic CNN:** The accuracy to be obtained by the basic network created from scratch is not that easy to estimate since it depends on a lot of variables. However it can be expected to get better than random results since it must learn something.

- **Transfer learning CNN's:** The accuracy to be obtained from these algorithms (FaceNet and VGG-Face) is expected to be fairly high since they are state-of-the-art networks, also trained on datasets very similar to the one created for this project. Here is a table that shows the resulting accuracy from applying FaceNet and some of its variants to the LFW dataset:

Model	Accuracy
nn4.small2.v1 (Default)	0.9292 ± 0.0134
nn4.small1.v1	0.9210 ± 0.0160
nn4.v2	0.9157 ± 0.0152
nn4.v1	0.7612 ± 0.0189
FaceNet Paper (Reference)	0.9963 ± 0.009

The FaceNet obtained an accuracy of 99.63%, this is almost perfect. From these results we can expect to obtain a very high accuracy by using these pre-trained networks.

III. Methodology

Data Preprocessing

During the data set creation, the pictures were cropped, as I stated previously, in order to keep only the faces. However, an extra processing step was applied to ensure the attainment of good results. This step consisted of the alignment of the faces in the pictures. Doing this as a sort of "normalization" of the input images. The alignment crops and scales the faces leaving the relevant face features near the same area in every image. This preprocessing step doesn't seem too affect much the data, however it helped to get the maximum potential from the pre-trained networks.

Implementation

The project was implemented on Python programming language. The main libraries used to implement the project were:

- Sklearn
- Keras
- facenet
- keras-vggface

Random Model

The random model was created by using the DummyClassifier provided by Sklearn library. The implementation doesn't require much effort nor parameters. The most important parameter for this random classifier is the *strategy* used for the random guessing. In this case I used the 'stratified' strategy, this seeks to get better than fully random results since this strategy takes into consideration the amount of training examples for each class and thus trying to guess the label by following the training data distribution.

One of the issues I had while implementing the random model was the fact that the classifier receives the data as a 1-dimensional vector and all the image inputs are read as a 3-dimensional matrix thus I had to do a quick pre-processing step of the input data to reshape it to the expected input dimension.

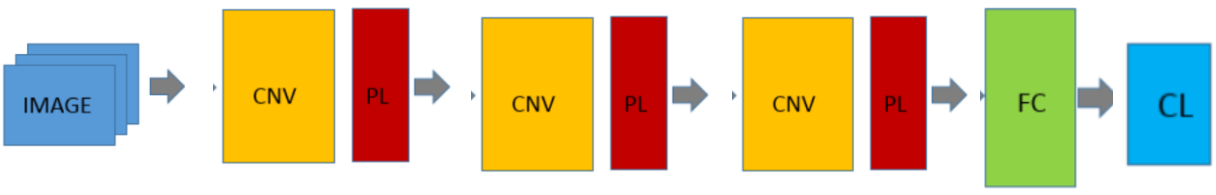
Here's some code:

```
def train_random_model():
    X_train = get_1D_image_features(train_files, IMAGE_SIZE)
    y_train = train_targets

    model = DummyClassifier(strategy='stratified',random_state=0)
    model.fit(X_train, y_train)

    return model
```

In this code we get the tensors from the images in the dataset (X_train, y_train) and we define then the DummyClassifier from sklearn (with strategy stratified as mentioned before) and finally we call the function fit to train the model and return it.

Basic Convolutional Network

This algorithm was built by using the Keras library which is an abstraction layer over other libraries like TensorFlow. This library allowed me to easily architect the network by specifying the intended layers to be used and their specific dimensions. Here's an image that shows the architecture of the final network used:



CNV = Convolutional Layer
PL = Pooling Layer
FC = Fully Connected Layer
CL = Classification Layer

This architecture is based on the common CNN's architectures to recognize images, it has 3 Convolutional-Pooling Layers one on top of the other, after that there is a Pooling Layer that applies the *average* operation, and finally is connected to a traditional fully connected layer and a classification layer in order to obtain the predicted class. After some trial-error this was the best performing architecture I obtained.

Here's some code:

```
def train_basic_net():
    train_tensors = get_tensors(train_files, IMAGE_SIZE)
    test_tensors = get_tensors(test_files, IMAGE_SIZE)

    model = Sequential()
    model.add(Conv2D(filters=16, kernel_size=3, padding='same', activation='relu', input_shape=(IMAGE_SIZE, IMAGE_SIZE, 3)))
    model.add(MaxPooling2D(pool_size=2))
    model.add(Conv2D(filters=32, kernel_size=3, padding='same', activation='relu'))
    model.add(MaxPooling2D(pool_size=2))
    model.add(Conv2D(filters=64, kernel_size=3, padding='same', activation='relu'))
    model.add(MaxPooling2D(pool_size=2))
    model.add(Flatten())
    model.add(Dense(256, activation='relu'))
    model.add(Dropout(0.3))
    model.add(Dense(80, activation='softmax'))

    # model.summary()

    if not os.path.isfile('../saved_models/weights.best.from_scratch_sub.hdf5'):
        checkpointer = ModelCheckpoint(filepath='../saved_models/weights.best.from_scratch_sub.hdf5',
                                       verbose=1, save_best_only=True)

        model.compile(optimizer='rmsprop', loss='categorical_crossentropy', metrics=['accuracy'])
        model.fit(train_tensors, train_targets,
                  epochs=EPOCHS,
                  validation_data=(test_tensors, test_targets),
                  batch_size=BATCH_SIZE,
                  callbacks=[checkpointer])

    model.load_weights('../saved_models/weights.best.from_scratch_sub.hdf5')

    return model
```

I separated the code into 3 main blocks to explain:

- In the first block, the traditional code to load the tensors from the input images. This code is shared between all the algorithms.
- In the second block is located the structure definition of the Convolutional Neural Network created by me. You can find in appearance order each of the layers described before. The code is really straightforward.
- In the third block you can find the conditional that searches for stored weights of the model to just load them and avoid re-training of the same network thus saving some time. If the model weights don't exists, then compile and train the model and then store the results with the usage of a *keras checkpointer* to make sure that we get the best model.

VGG-Face convolutional network

To implement this algorithm I had to work a lot, because I had to follow the complete steps for doing transfer learning:

1. Load the network and its weights
2. Obtain the convolutional features by applying the pre-trained network to all the input data
3. Train a classifier to get the convolutional features and labels as input and classify the data.

I used a library named *keras-vggface* which had the implementation of the network and a set of weights obtained by trained the model on the VGGFace dataset. The usage of the network is pretty straightforward however I encountered some problems during my implementations:

- Applying the network to the whole dataset takes a lot of time (around 2 hours) in order to get the convolutional features needed to train the final classifier and get the results. While I was writing code I couldn't wait 2 hours each time I wanted to test or fix something, so I had to implement a function to store the convolutional features obtained from VGGFaceNetwork to speed up the process.
- Another encountered problem was the fact that the net wasn't performing well, and after some research I found that the images needed a special pre-processing step to remove the mean pixel in each color channel of the image. This way the algorithm will be given the input data in the same format that the one on which it originally trained.

Here's some code:

```
def save_vggface_bottleneck_features():
    print("Getting bottleneck features for VGGFace")

    # build the VGG16 network
    model = VGGFace(include_top=False, input_shape=(224, 224, 3), pooling='avg')

    train_tensors = get_tensors_for_vggface(train_files, IMAGE_SIZE)
    bottleneck_features_train = [model.predict(np.expand_dims(tensor, axis=0)) for tensor in tqdm(train_tensors)]
    len(bottleneck_features_train)
    np.save(open('../bottleneck_features/vggface_train_2.npy', 'w'), bottleneck_features_train)

    test_tensors = get_tensors_for_vggface(test_files, IMAGE_SIZE)
    bottleneck_features_test = [model.predict(np.expand_dims(tensor, axis=0)) for tensor in tqdm(test_tensors)]
    np.save(open('../bottleneck_features/vggface_test_2.npy', 'w'), bottleneck_features_test)

    valid_tensors = get_tensors_for_vggface(valid_files, IMAGE_SIZE)
    bottleneck_features_validation = [model.predict(np.expand_dims(tensor, axis=0)) for tensor in tqdm(valid_tensors)]
    np.save(open('../bottleneck_features/vggface_valid.npy', 'w'), bottleneck_features_validation)
```

This first code snippet shows the function used to obtain the convolutional features from the VGG-Face network. It load the pre-trained net and the applies them (by making predictions) to the train/test/valid datasets, finally storing them in the corresponding files for future usage.

```
def train_vggface_net():
    if not (os.path.isfile('../bottleneck_features/vggface_train.npy') and
        os.path.isfile('../bottleneck_features/vggface_test.npy') and
        os.path.isfile('../bottleneck_features/vggface_valid.npy')):
        save_vggface_bottleneck_features()

    train_data = np.load(open('../bottleneck_features/vggface_train.npy'))
    test_data = np.load(open('../bottleneck_features/vggface_test.npy'))

    model = Sequential()
    model.add(Flatten(input_shape=train_data.shape[1:]))
    model.add(Dense(512, activation='relu'))
    model.add(Dropout(0.3))
    model.add(Dense(80, activation='softmax'))

    # TRAIN ONLY IF NEEDED
    if not os.path.isfile('../saved_models/weights.best.from_vggface.hdf5'):
        checkpointer = ModelCheckpoint(filepath='../saved_models/weights.best.from_vggface.hdf5',
                                       verbose=1, save_best_only=True)

        model.compile(optimizer='rmsprop', loss='categorical_crossentropy', metrics=['accuracy'])
        model.fit(train_data, train_targets,
                  epochs=EPOCHS,
                  batch_size=BATCH_SIZE,
                  validation_data=(test_data, test_targets),
                  callbacks=[checkpointer])

    model.load_weights('../saved_models/weights.best.from_vggface.hdf5')

    return model
```

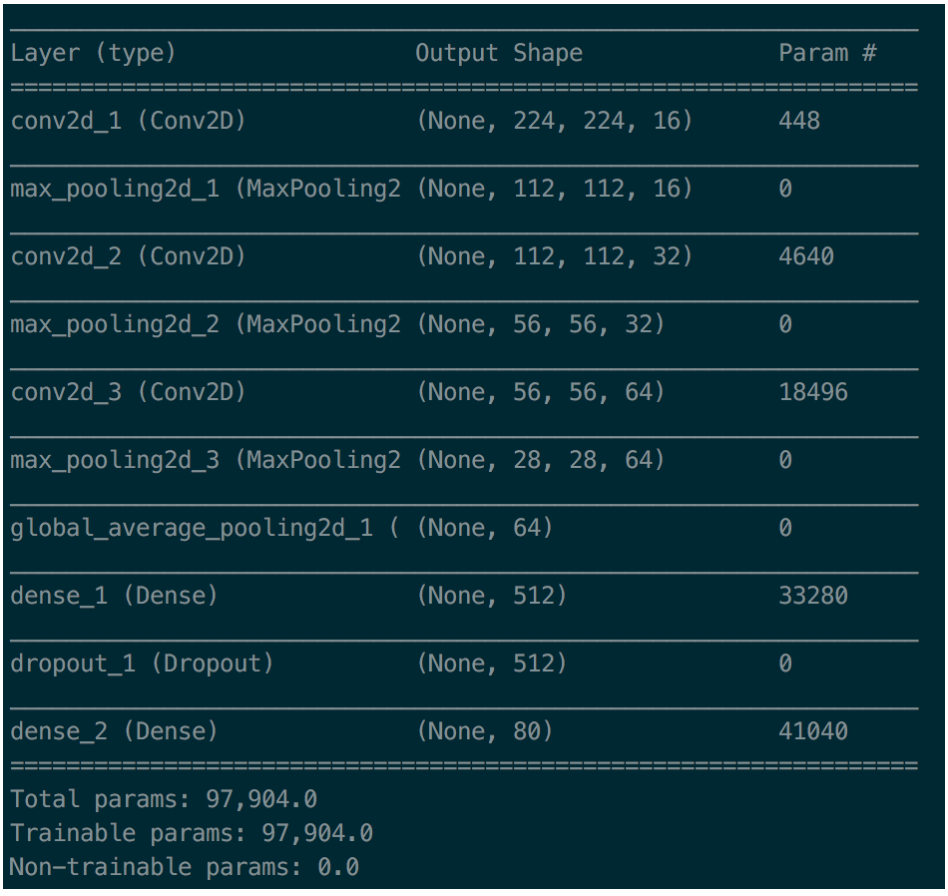
This second code snippet is the one in charge for training the VGG-Face last layers. In the first part of the code it validates that the convolutional features have been extracted from the dataset and if not it asks for its calculation, then the convolutional features for the train and test set are loaded. After the data is loaded the final layer of the network is trained, in this case there is a Flatten layer to vectorize the input and pass it to a Fully Connected layer with 512 neurons and finally the activation layer. This 2 layers are trained passing them as input the convolutional features and then the results are stores to avoid retrain the same way as with the BasicNet.

With the convolutional neural networks there are a lot of parameters that can be modified to get different results. The changes can come in the form of an architecture and in the form of layers parameters, activation functions, metrics to optimize and the optimizer algorithm. In this case I used the following parameters:

- **Optimizer:** The optimizer I used is one of the more robust existing (RMSprop). This optimizer behaves really well since it includes important concepts like momentum, gradient normalization and adaptable step rates.
- **Loss function:** The function I used was Categorical crossentropy which is one of the classics too for multi-class classification problems.

The experiments I did were more related to the network architecture than the parameters, since these tend to work well, I tried different number of convolutional layers and with different depths. Also tried different pooling layers (max, average) and finally I tested with different number of fully connected layers and their number of neurons. After some experimenting, I finally achieved the best accuracy using the following composition:

Figure 8



Layer (type)	Output Shape	Param #
conv2d_1 (Conv2D)	(None, 224, 224, 16)	448
max_pooling2d_1 (MaxPooling2	(None, 112, 112, 16)	0
conv2d_2 (Conv2D)	(None, 112, 112, 32)	4640
max_pooling2d_2 (MaxPooling2	(None, 56, 56, 32)	0
conv2d_3 (Conv2D)	(None, 56, 56, 64)	18496
max_pooling2d_3 (MaxPooling2	(None, 28, 28, 64)	0
global_average_pooling2d_1 (	(None, 64)	0
dense_1 (Dense)	(None, 512)	33280
dropout_1 (Dropout)	(None, 512)	0
dense_2 (Dense)	(None, 80)	41040

Total params: 97,904.0
Trainable params: 97,904.0
Non-trainable params: 0.0

With the VGG-Face network I have been able to test different Fully Connected layers size for the final classification part, however all the results were the same since the accuracy is really high (almost perfect). Here are some experimentes and their results:

Experiment	Accuracy (Validation set)
Experiment 1	9.76%
Experiment 2	10.02%
Experiment 3	12.22%
Experiment 4	12.56%

- **Experiment 1**: In this experiment a simple Flatten layer was used between the Convolutions and the final neural network for prediction. A number of 256 neurons were used inside the fully connected layer.
- **Experiment 2:** In this experiment a Max pooling layer was used between the Convolutions and the final neural network for prediction. A number of 256 neurons were used inside the fully connected layer.
- **Experiment 3:** In this experiment a simple Global average layer was used between the Convolutions and the final neural network for prediction. A number of 256 neurons were used inside the fully connected layer.
- **Experiment 4:** In this experiment a simple Global average layer was used between the Convolutions and the final neural network for prediction. A number of 512 neurons were used inside the fully connected layer.

IV. Results

Model Evaluation and Validation

The algorithms were presented with the training and testing set during training phase in order to learn. After experimenting with the different algorithms, I have got the following accuracy results by applying the 4 models to a new unknown set of data (validation set):

Models	Accuracy (Validation set)
Random	0.75%
BasicNet	12.56%
VGG-Face	98.24%
FaceNet	98.74%

The best models to solve the problem, as expected, were the two state-of-the-art pre-trained networks. The generalization power of these two networks was used to get a vector of features for each image and then passed to classifiers to get the result.

The only param I modified while using the pre-trained networks was the number of existing neurons in the final fully connected layer, used to learn the weights for each of the obtained convolutional features from the VGG-Face model to the class label. In this case I used 512 neurons, one for each feature.

The transfer learning technique is a very powerful tool and it has proven its potential by obtaining such great results. This models would be definitely the way to go if I had to implement some application involving face recognition.

In order to test the robustness of the models, I tested them against the validation set raw (without any preprocessing) and I got the following results:

Models	Accuracy (Raw Validation set)
Random	1.00%
BasicNet	10.75%
VGG-Face	98.50%
FaceNet	98.27%

It can be noticed that the performance of the chosen models doesn't vary from the original, the accuracy remains almost the same. This means that the algorithms are capable of generalizing well, ensuring this way good results when applied to unseen data.

Justification

Earlier in the project, two results were proposed as a benchmark for the results of the algorithms:

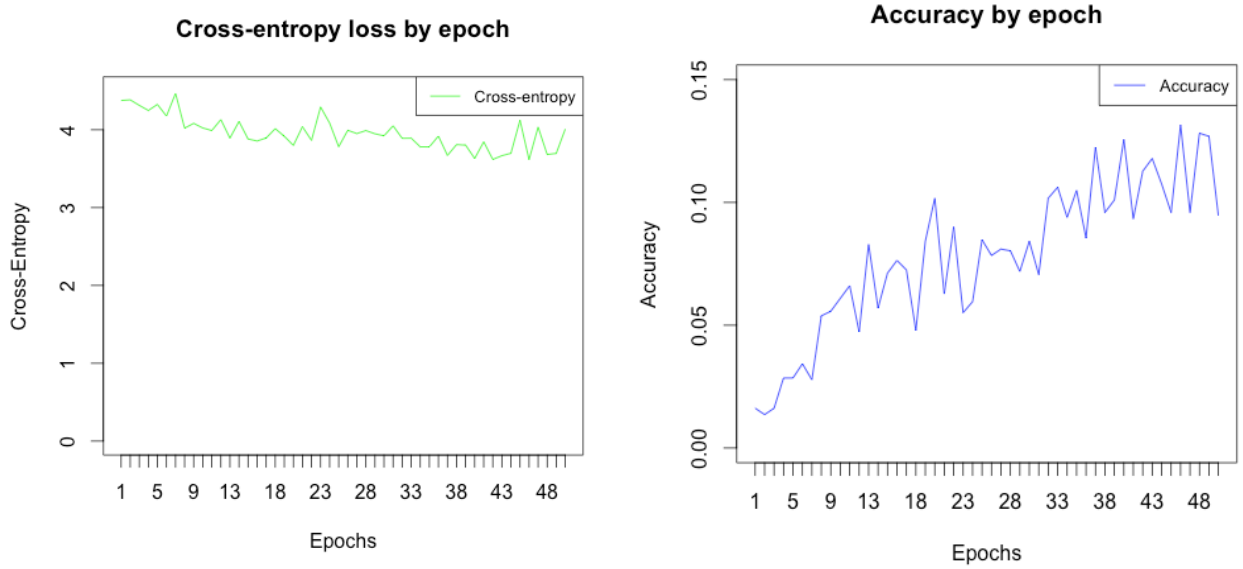
- The first benchmark metric was the random classifier expected results. The 3 non-random models were expected to get better than random results in terms of accuracy, this means more than 1.25% accuracy. This goal was clearly achieved by all of the three models. The two state-of-the-art models achieved almost perfect score while the BasicNet model obtained 10 times more accuracy than random guessing.
- The second benchmark metric was the accuracy obtained by the FaceNet model on the LFW dataset. In this case the only two competitors are my VGG-Face and FaceNet models, and although they achieved more than 98% accuracy, they weren't able to break the 99% barrier.

The FaceNet and VGG-Face models are clearly capable of delivering great results in terms of accuracy when it comes to the facial recognition of the 80 celebrities present in my custom dataset. An interesting fact to keep in mind is that the humans usually have a 97% accuracy on average on this task. This means that the models are able to perform like humans and even a bit better.

V. Conclusions

Free-Form Visualization

The multi-class classification problem is a hard task, and even harder when it comes to the facial recognition task. An interesting thing that I've found during the implementation of the project is that the rate of improvement of the convolutional networks is really slow. They need a lot of time to train, in my case I trained the BasicNet model for about 5 hours each time that a parameter was changed. A lot of minutes were required for each training epoch and the improvement of the net is not that fast. In the following graph, we can see the evolution of the cross-entropy loss (the metric to optimize by the classifier) and the accuracy on the test set (the metric used to measure the quality of the model):



Reflection

I have passed through a lot of phases in order to solve the selected problem, these can be summarized as follows:

1. Analyze the input of the problems to find any necessary additional pre-processing step to be applied, in this case I had to align the images and resize them to match expected inputs for some of the pre-trained models.
2. Separate the data into 3 different sets:
   - **Training**: this is the set to be used for the algorithm to learn
   - **Test:** this is the set to validate during the training set that the algorithm is moving in the right direction
   - **Validation:** this is the hidden set to be used to test the accuracy on the resulting model.
3. Implement the chosen models, in this case 4 models were implemented to compare:
   - Random Classifier
   - Basic Convolutional Network (BasicNet)
   - VGG-Face

    - FaceNet

    In the case of the BasicNet, a few different architectures were tried to improve accuracy.
4. Test the models and measure their final accuracies.
5. Compare the models and analyze the obtained results.

During the implementation of the project I have found two of the phases of the application of Transfer Learning technique particularly challenging:

- The phase of preprocessing for the images to be passed to the VGG-Face algorithm, this task took a while to get right since the library documentation is not so clear about this, they even have a bug in their example when applying the subtraction of the mean pixel for each of the color channels of the picture. I also had to read around Kaggle forums to understand that the common pixel normalization (divide each pixel by 255) is not necessary in the case of the VGG-Face network.

- The phase of getting the convolutional features from VGG-Face in order to pass them to a mini neural network to get the final class. It wasn't so clear to me that this step had to be done for each training/test/validation sample, so I had to implement a function to apply the pre-trained network to each of the instance in each set and then store them in a file to speed up the following executions of the code and to test faster.

The models I implemented were all interesting and I'm really satisfied because I had the opportunity of testing two state-of-the-art models on my custom dataset.

Improvement

Although I had the opportunity of running state-of-the-art models on my dataset, and considering that my dataset is smaller and simpler that the ones used to train and test originally these models, I wasn't able to get accuracies above the 99%. And probably the way to achieve such results is by using another technique named Fine-Tuning. This technique consists on freeze some of the layers inside of the pre-trained models and retrain slightly the rest of the layers to make the convolutional networks more familiar with the used dataset. Additionally to this, some of the usual training parameters could be modified during this re-training phase, all of this translate to better results potentially. In my case I wasn't able to apply this, because a lot of computational power (GPU's) and time is required to re-train the networks and I don't have the resources.

References

[1] T. Kohonen. Self-organization and Associative Memory. Springer-Verlag, Berlin, 1989.

[2] P. J. B. Hancock and L. S Smith. *GANNET: Genetic design of a neural net for face recognition. In H-P. Schwefel and R. Manner*, editors, Proceedings of the Conference on Parallel Problem Solving from Nature. Springer Verlag, 1991.

[3] L. Sirovich and M. Kirby. *A Low-Dimensional Procedure for the Characterization of Human Faces*, J. Optical Soc. Am. A, 1987, Vol. 4, No.3, 519-524.

[4] Florian Schroff, Dmitry Kalenichenko and James Philbin. *FaceNet: A Unified Embedding for Face Recognition and Clustering*. 2015

[5] Viola-Jones Algorithm.
https://en.wikipedia.org/wiki/Viola%E2%80%93Jones_object_detection_framework

[6] FaceNet: A Unified Embedding for Face Recognition and Clustering.
https://arxiv.org/abs/1503.03832

[7] Deep Face Recognition.
http://www.robots.ox.ac.uk/~vgg/publications/2015/Parkhi15/parkhi15.pdf