

INTERIM REPORT

IMPERIAL COLLEGE LONDON

DEPARTMENT OF COMPUTING

Deep Learning For Facial Expression Analysis

Author:

Tom Bartissol (CID: 00824562)

Date: June 20, 2017

Contents

1	Introduction	4
2	Background	5
2.1	Facial Expressions	5
2.1.1	Facial Action Coding System (FACS)	5
2.1.2	Interpreting emotions	5
2.2	Facial Expression Analysis	7
2.3	Databases	7
2.4	Deep Learning	8
2.4.1	Deep Convolutional Neural Networks	9
2.4.2	Activation functions	13
2.4.3	Loss functions	16
2.4.4	Optimisers	17
2.4.5	Models	20
2.4.6	Evaluation Measures	22
3	EmotioNet Database	24
3.1	Downloading	24
3.1.1	Reading the xlsx file	24
3.1.2	Downloading the image	25
3.1.3	Storing the image	26
3.2	Converting to TFRecords	26

3.3	Extending TF-Slim datasets	27
3.4	Annotating Valence and Arousal	28
4	Action Unit Prediction	28
4.1	Adapting training scripts	28
4.1.1	Duplicating the Logits	28
4.1.2	Cleaning the Labels	29
4.1.3	Changing the final activation function	29
4.2	Prediction of 60 Action Units	30
4.2.1	Fine-tuning	30
4.2.2	Evaluation	31
5	Valence and Arousal Regression	31
5.1	VGG 16	31
5.2	ResNet	31
6	Future Work	31

1 Introduction

The ability to analyse human facial expressions is an active area of research with exciting near-future real-world applications ranging from law enforcement to advertising (measuring how positively or negatively people respond to an ad). It would also be at the core of any system capable of intelligent Human-Computer Interaction (HCI). Indeed, for such interactions to be life-like, the Computer should be able to recognise human emotions which are expressed through multiple channels, an important one being facial expressions. *Interpreting* the recognised facial expressions into the six *basic emotions* [2] (see Table 2) would allow the Computer to identify human emotions to some extent.

Facial Expressions result from the contraction of a facial muscle or a group of facial muscles and the visually perceptible changes due to such contractions have been codified by Ekman and Friesen into the well-known Facial Action Coding System (FACS) [3] which provides labels, facial *Action Units* (AUs), for the actions of a muscle or group of muscles. For instance, AU 12 corresponds the lip corner puller and if it is active at the same time as AU 6, which corresponds to the cheek raiser, then the subject is smiling and this could be interpreted as happiness.

We are therefore concerned with two problems (i) identifying these AUs and (ii) estimating continuous emotion dimensions: valence (how positive or negative the emotion is) and arousal (how intense) instead of discrete emotions such as the six basic emotions. Since changes in muscular activity are not instant but last between 250ms and 5s [4], we are interested in data with a temporal dimension, that is videos. More specifically, whereas the majority of previous work has been conducted on data sets captured in constrained environment and/or using acted or posed facial expressions [9], we are interested in the so called "*in-the-wild*" videos, which are recorded in different lighting conditions with different poses and most accurately represent spontaneous facial expressions.

To achieve this, we will use end-to-end deep learning models, namely ResNet, VGG and ResNet-Inception. Each model will be trained/fine tuned to jointly solve problems (i) and (ii) on data taken in-the-wild. Finally we will evaluate these models on existing and established benchmarks and databases (not necessarily in-the-wild).

2 Background

2.1 Facial Expressions

2.1.1 Facial Action Coding System (FACS)

The human head having a finite number of muscles, the number of visually perceptible changes that can be caused by the contraction or relaxation of one or more of these muscles, that is the number of facial expressions, is finite. We can therefore taxonomise these facial changes into a coding system. Although there exists a few such coding systems [7], the most popular and used coding system is the FACS [3]. FACS breaks down each visually perceptible change into facial *Action Units* (AUs) which roughly corresponds to the contraction or relaxation of individual facial muscles. The activation of one or more AU creates a facial expression. Some examples of AUs and their associated facial are listed in Figure 1.

On top of this taxonomy, FACS also provides an intensity score, A-E (maximum-minimum), to rate how pronounced each AU is in a facial expression (see Table 1). For instance AU 12A would indicate that the lip corners are slightly pulled whereas AU 12E would indicate that they are maximally pulled.

Finally, AUs do not activate instantly. Indeed, an AU is firstly in a neutral state, then, when it starts activating, muscles contract and the AU is said to be in an *onset* phase, once the muscles are contracted and the AU is at its peak, it is said to be in an *apex* phase, finally, when the muscles start to relax and the AU starts to disappear, the AU is said to be in an *offset* phase before returning to normal. So the activation of an AU generally follows the following pattern: neutral - onset - apex - offset - neutral.

2.1.2 Interpreting emotions

2.1.2.1 Discrete case Emotions can roughly be broken down into six *basic emotions* [2], seven if we count the neutral emotion. We can then interpret a facial expression into one of these seven categories. Note that this is only an interpreta-

Intensity	description
A	Trace
B	Slight
C	Marked or Pronounced
D	Severe or Extreme
E	Maximal

Table 1: The scale for measuring the intensity with which an AU is activated

tion as emotions are expressed though multiple channels such as body language or voice and facial expressions are only one of these channels.

2.1.2.2 Continuous case This is the case in which we are interested. Emotions can be represented using two continuous dimensions:

- **Valence:** characterises the attractiveness or aversiveness of an emotion, that is respectfully how positive or negative the emotion is.
- **Arousal:** characterises the intensity of the emotion

We can therefore represent emotions in a two dimensional coordinate system using their valence and arousal values, see Figure 2

Emotion	AUs
Neutral	0
Happiness	6, 12
Sadness	1, 4, 14
Surprise	1, 2, 5B, 26
Fear	1, 2, 4, 5, 7, 20, 26
Anger	4, 5, 7, 23
Disgust	9, 15, 16

Table 2: The 6 basic emotions (plus neutral) and the corresponding Action Units that are generally activate when the emotion is present


























Upper Face Action Units					
AU 1	AU 2	AU 4	AU 5	AU 6	AU 7
					
Inner Brow Raiser	Outer Brow Raiser	Brow Lowerer	Upper Lid Raiser	Cheek Raiser	Lid Tightener
*AU 41	*AU 42	*AU 43	AU 44	AU 45	AU 46
					
Lid Droop	Slit	Eyes Closed	Squint	Blink	Wink
Lower Face Action Units					
AU 9	AU 10	AU 11	AU 12	AU 13	AU 14
					
Nose Wrinkler	Upper Lip Raiser	Nasolabial Deepener	Lip Corner Puller	Cheek Puffer	Dimpler
AU 15	AU 16	AU 17	AU 18	AU 20	AU 22
					
Lip Corner Depressor	Lower Lip Depressor	Chin Raiser	Lip Pucker	Lip Stretcher	Lip Funneler
AU 23	AU 24	*AU 25	*AU 26	*AU 27	AU 28
					
Lip Tightener	Lip Pressor	Lips Part	Jaw Drop	Mouth Stretch	Lip Suck

Figure 1: Facial Action Units

2.2 Facial Expression Analysis

In the early days, facial expression analysis was restricted to recognising the six basic emotions. Furthermore, computer vision techniques were used to extract features from input images and then classify them instead of using the *end-to-end* deep learning techniques we will use in this project. As indicated in [9], the interested reader is directed towards [5, 6] for a thorough overview of these early techniques.

2.3 Databases

We direct the reader towards [9] for a complete survey of the main databases available for facial expression analysis. We are interested in databases containing images or videos taken in-the-wild and annotated with facial action units. As such, we will use the EmotionNet [1] database which contains 1,000,000 in-the-wild images of emotions. These images were downloaded from the Internet by using specific combinations of keywords, such as “feeling angry” or “feeling happy”, and were then

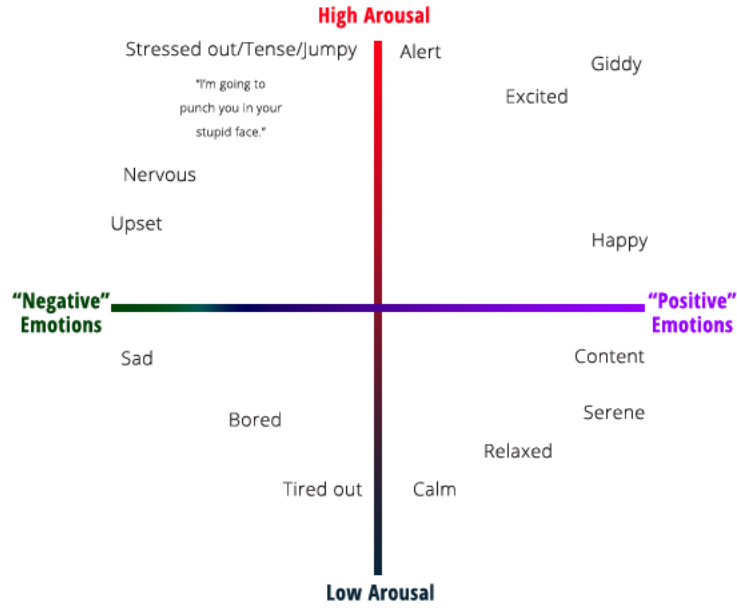


Figure 2: Plotting emotions according to their valence and arousal

automatically annotated with AUs and AU intensities as well as emotion categories by using a novel computer vision algorithm presented in [1].

However, we are interested in a subset of 25,000 images from this database that were manually annotated as we consider the accuracy of these annotations to be higher than those produced by the algorithm mentioned above. This subset was annotated for Action Units activation with '1' meaning that the action unit is active, '0' that is not active and '999' that the action unit is occluded or that it has not been annotated. An example image from this dataset is presented in Fig. 3

2.4 Deep Learning

We use deep learning models in order to, given an image of a facial expression, recognise which AUs are active and which aren't as well as estimate the valence and arousal of the given facial expression. We use these models as they have proved to be able to learn to represent abstract data features (e.g a smile) by using multiple computational layers.

More formally, a deep learning model is a neural network with a set of n



Figure 3: Example image from the EmotioNet database. Only two AUs, 6 and 12, are labelled as active (i.e. '1'). AUs 1, 2, 4, 5, 9, 17, 20, 25, and 26 are labelled as inactive and the remaining AUs are labelled as occluded or '999'

layers. Let a_j^i be the output, or activation, of the j^{th} neuron of the i^{th} layer for $i = 1 \dots n$ where the case $i = 1$ corresponds to the input vector. Then we can relate a_j^i to the outputs of the previous layer as follows:

$$a_j^i = f(z_j^i) \quad (1)$$

$$z_j^i = s_j^i + b_j^i \quad (2)$$

Where s_j^i is the some weighted sum of all or part of the activations a_j^{i-1} for $j = 1 \dots m_{i-1}$ (where m_{i-1} is the number of neurons in layer $i - 1$) of the previous layer, b_j^i is a bias term and $f(\cdot)$ is an activation function which ensures non linearity.

2.4.1 Deep Convolutional Neural Networks

In particular, since our task focuses on images (i.e. 2/3-dimensional data), we will be using a particular type of deep learning models called deep convolutional neural networks (DCNNs). DCNNs consist of a succession of convolutional and pooling layers followed by fully connected layers.

The first block of layers, the convolutional and pooling layers, act as a feature extractor. They take as input a raster image $\in \mathcal{R}^{n \times m}$ where n is the image height and m the image width, either in greyscale ($\in \mathcal{R}^{1 \times n \times m}$) or in RGB colours $\in \mathcal{R}^{3 \times n \times m}$

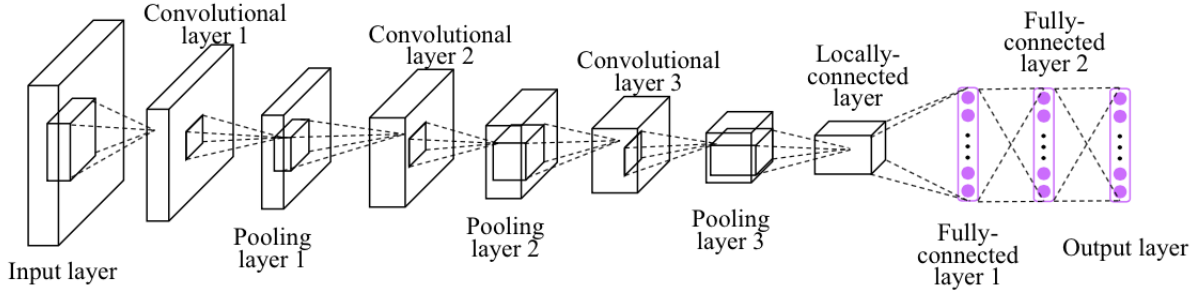


Figure 4: example of a DCNN architecture with a succession of convolutional + max pooling layers followed by a bloc of fully connected layers

Source: http://personal.ie.cuhk.edu.hk/~ccloy/project_target_code/images/fig3.png

(where 3 is the depth of the image, i.e. the number of channels) and processes it to extract features which will constitute a meaningful representation of the image in a lower dimension, i.e. a vector.

2.4.1.1 Convolutional Layers Convolutional layers are the core components of DCNNs. They are where the most heavy computations take place and act as feature extractors by applying a set of spatially small learnable kernel filters to the input image. These filters could for instance have a size of $7 \times 7 \times D$ where the first and the second 7 correspond to the height and the width of the filter respectively and D corresponds to the depth (either 1 or 3 if we are working with RGB input images).

Each filter is applied by “sliding” it over the input image and performing a convolution at every pixel of the image between the filter and the portion of the image that lies directly beneath it. Mathematically, this gives the following: suppose we have an input image $I \in \mathcal{R}^{W \times H}$ and a filter $F \in \mathcal{R}^{k \times k}$ such that $I(a, b)$ denotes the pixel on row a and column b of I (assume the same notation for F). Then the convolution operation $C(.,.)$ executed at pixel (a, b) of the image is

$$C(a, b) = \sum_{i=1}^k \sum_{j=1}^k F(i, j) \times I(a-i, b-j) \quad (3)$$

small

Therefore, as we slide a filter over the image, we produce a 2D response

map that corresponds to the convolved features at each spatial location of the input image, see Fig. 5. Intuitively, each kernel filter learns to detect specific features in the input such as a corner or a blob of colour

In the case of a 3 dimensional input volume ($Width \times Height \times Depth$), each filter will also be 3 dimensional with the depth dimension D matching that of the input. Therefore each filter will produce D response maps which are summed together with a bias term to produce a 2 dimensional overall response map for that filter.

Finally, when using a $k \times k$ kernel on an $n \times m$ image, the response map will have a smaller size of $(n - l) \times (m - l)$ where $l = \frac{k-1}{2}$ (k is usually an odd number). While we do want to perform dimensionality reduction before going through the fully connected layers, this means that the kernel might omit some important features that are at the edges of the image. To solve this problem, it is common practice to add l layers of zero-padding on each side of the image so that the kernel also covers the edges of the image and the response map has the same dimension as the input.

To sum up, here are the inputs and the outputs of a typical convolutional layer with N kernel features of size $k \times k$ applied using a stride of s on an image with a padding of p :

1. **In:** an image with size $W \times H \times D$ where W is the width of the image, H the height and D the depth
2. **Out:** a volume with size $W_{out} \times H_{out} \times N$ where $W_{out} = \frac{W-k+2p}{s}$, $H_{out} = \frac{H-k+2p}{s}$ and N is the number of filters used at that layer

2.4.1.2 Pooling Layers As mentioned in the above paragraph, convolutional layers can perform dimensionality reduction if no zero-padding is added to the input image. However this job generally falls to pooling layers.

As their name suggests, pooling layers group pixels together to reduce them to a single pixel. There are several ways to determine the value of the output pixel and the most common one is maximum pooling which consists in outputting the

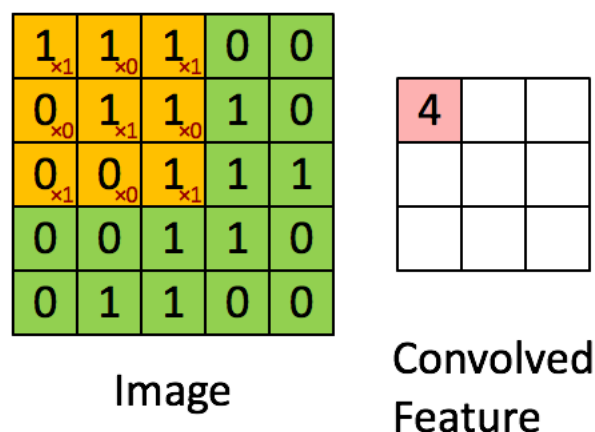


Figure 5: Example of a convolution operation at a single spatial location. The orange square represents the kernel filter. Note that no padding has been applied here which is why the convolved feature (i.e. the response map) is smaller.

Source: http://deeplearning.stanford.edu/wiki/images/6/6c/Convolution_schematic.gif

pixel in the group that has the highest value. Since this is just a static operation, there are no learnable parameters involved. The only parameters are the spacial extent p (i.e. the width and height) of the pooling region/filter and the stride s which defines how many pixels to skip until applying the next pooling operation. For instance, a pooling layer with $p = 2$ (2x2 filters) and $s = 2$ will divide the width and height of the input image by 2 (it does not change the depth) as can be seen in Fig 6.

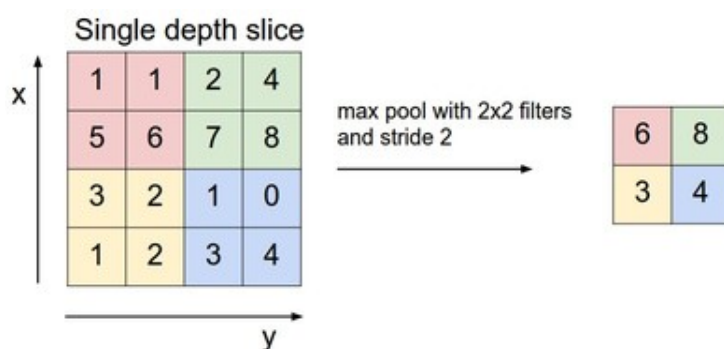


Figure 6: example of a single max pooling operation with a 2×2 pooling filter and a stride of 2

Source: <http://cs231n.github.io/assets/cnn/maxpool.jpeg>

To sum up here are the inputs and outputs of a pooling layer:

1. **In:** k images of size $n \times m \times d$ where n is the width, m the height and d the depth of the image
2. **Out:** k responses of size $\left(\frac{n-p}{s} + 1\right) \times \left(\frac{m-p}{s} + 1\right) \times d$ where p is the width of the square pooling filter and s the stride size

2.4.1.3 Fully Connected Layers Fully connected layers constitute the second block of the

Unlike convolutional layers, which are locally connected (each pixel in the response map is locally connected to a sub-region of the input), fully connected layers have their outputs connected to *every* input.

As such, the activations can be simply calculated via matrix multiplication between the inputs and their associated weights and adding a bias term before passing all that to an activation function (e.g. ReLu, Sigmoid, Softmax).

It is worth noting that we can easily replace a fully connected layer by an equivalent convolutional layer; all that must be done is to match the size of the kernel filters to the size of the input image or vector. For instance, in the VGG 16[CITE] network, the final convolutional layer outputs 512 feature maps of size 7×7 which are passed on to a fully connected layer with 4096 neurons. Then we can replace this fully connected layer by a convolutional one with 4096 filters of size 7×7

2.4.1.4 Dropout Layers Dropout layers[CITE] are a simple yet powerful regularisation technique to prevent, overfitting. The key idea is to randomly block some neurons from connecting to the next layer during training see Fig. 7. This prevents groups of neurons from co-adapting too much and improves the flexibility of the network which decreases overfitting.

2.4.2 Activation functions

Each trainable layer in the network also has an activation function, through which it passes its outputs before sending them to the next layer. The main role of activation

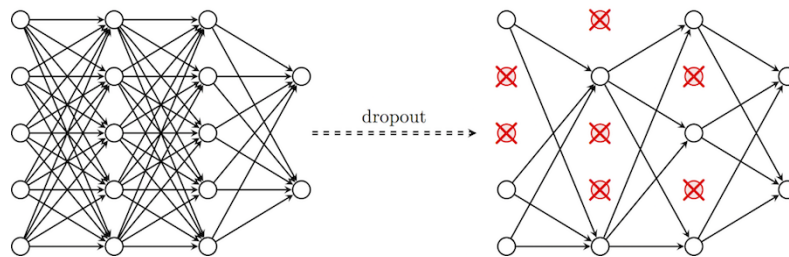


Figure 7: example dropout layers: the crossed neurons are blocked from sending their inputs to the next layer

Source: <https://cambridgespark.com/content/tutorials/convolutional-neural-networks-with-keras/figures/drop.png>

functions is to provide non-linearity to the network. Intuitively, this enables deep networks to better fulfill their representation learning power. Indeed, without these functions, deep networks (or any neural network) would only be learning a linear combination of the input data which is quite limiting. We now go through the main activation functions used in neural networks.

2.4.2.1 Sigmoid

The sigmoid activation function is a special case of the logistic function and is defined as follows:

$$f(x) = \frac{1}{1 + e^{-x}} \quad (4)$$

It was a popular activation function until the arrival of the ReLU (2.4.2.3) which dealt better for the main shortcoming of this one. Indeed, the sigmoid function squashes everything between 0 and 1 therefore, as the absolute value of the input increases, the gradient at that point decreases. So the gradient tends towards 0 as the input tends towards infinity. This is called the vanishing gradient problem and is a problem as when performing backpropagation, the updates will not change the weights significantly which considerably slows training at best or prevents us to reach a local minimum at worst.

Today, the sigmoid activation function, just like softmax (see paragraph

below) is used on the very last layer of the network, to make the outputs look like probabilities.

2.4.2.2 Softmax

Also known as the Normalised Exponential, the softmax activation function is defined as follows: consider the i^{th} layer of a neural network and let \mathbf{z}^i be the vector of raw activations of this layer as defined in (2), then

$$a_j^i = f(\mathbf{z}^i)_j = \frac{e^{z_j^i}}{\sum_{k=1}^{m_i} e^{z_k^i}} \quad (5)$$

Where m_i denotes the number of neurons in layer i .

From a probability theory standpoint, the softmax activation function calculates a probability distribution over what we can consider as m_i categories. It is therefore mostly used on the very last layer of neural networks trained to classify inputs into a single category. Therefore when using a softmax activation function on the last layer, one assumes that these categories are mutually exclusive, meaning that training or test instance can not belong to two or more categories at the same time.

2.4.2.3 ReLU

In its simplest form, the Rectified Linear Unit[CITE] for a scalar input x is defined as follows:

$$f(x) = \max(0, x) \quad (6)$$

Variations have recently emerged such as the Leaky ReLU[CITE] which is defined as:

$$f(x) = \begin{cases} x, & \text{if } x > 0 \\ 0.01x, & \text{otherwise} \end{cases} \quad (7)$$

There are two main advantages to using ReLUs which make them the most popular activation functions today. The first is that it significantly reduces the likelihood of the gradient to vanish (or be extremely small) since when $x > 0$, the gradient has a constant value, unlike the sigmoid activation function described above whose gradient becomes increasingly small as x increases. This makes learning significantly faster. The second advantage is that using ReLU gives more sparse representations of the data as any negative input is mapped to 0. Sparse representations seem to be more efficient as they encode a *de facto* feature selection (if an input is mapped to 0 then that “feature” is ignored) and therefore have more potential to uncover true relationships in the data.

2.4.3 Loss functions

Loss functions are at the core of deep neural networks as they define the objective where are trying to achieve. It is therefore crucial to select the the appropriate loss function depending on the task performed by the network.

2.4.3.1 Mean Squared Error

The Mean Squared Error is the most widely used loss function for regression problems. For a data set $(\mathbf{x}_i, y_i)_{i=1}^N$ with N examples, it is defined as:

$$MSE = \frac{1}{2N} \sum_{i=1}^N (y_i - \hat{y}_i)^2 \quad (8)$$

Where \hat{y} is the network’s prediction using input \mathbf{x}_i .

2.4.3.2 Cross Entropy

The cross entropy loss function is generally used in classification tasks, when the network outputs a probability distribution over the classes in which an

example can be categorised, either by using the softmax (2.4.2.2) or the sigmoid (2.4.2.1) activation function.

Suppose the network outputs a probability distribution \hat{y}^i for m categories and the target probability distribution is y^i for $i = 1 \dots N$ examples. Then the cross entropy loss for a single example is defined as:

$$H(y^i, \hat{y}^i) = -\frac{1}{m} \sum_{k=1}^m y_k^i \log(\hat{y}_k^i) \quad (9)$$

And the cross entropy loss of the whole data set is just the average of the losses of each example in the data set as defined in Eq. (9).

2.4.4 Optimisers

Now that we have overviewed the different elements that make up the structure of a network, we must look at how to update the trainable parameters (i.e. the weights). The key value that we use to update a parameter is the gradient of the loss function (2.4.3) with respect to that parameter to perform gradient descent. This gradient, Δw , is computed using the backpropagation[CITE] algorithm and the most simple way to apply it to update the parameters w is as follows:

$$w \leftarrow w - \alpha \Delta w \quad (10)$$

where α , the learning rate, controls the magnitude of the updates. In this setting, Δw is calculated after a forward pass of the entire training data set. However this is computationally inefficient or unfeasible so we use different techniques.

2.4.4.1 Stochastic Gradient Descent

The first technique, Stochastic Gradient Descent (SGD), computes the gradient Δw_{batch} w.r.t. the parameters after a forward pass of a small batch (x_{batch}, y_{batch})

of training examples, usually 128 or 256, instead of the entire dataset. It then updates the parameters as in Eq. (10):

$$\mathbf{w} \leftarrow \mathbf{w} - \alpha \Delta \mathbf{w}_{batch} \quad (11)$$

Since the gradients are only calculated from a small sample of the dataset, the variance of the updates between each batch will be higher, which is why we typically use a small learning rate of $\alpha = 0.1$ to $\alpha = 0.001$.

This technique is stochastic because we randomly shuffle the data before dividing it into batches and calculating and applying the updates. Ideally, this should be done before each epoch, where an epoch is defined as a pass over the entire data set.

2.4.4.2 SGD with Momentum

The objective of the parameter updates is to descend the gradient of the loss function to find a good local minimum. Viewing this gradient as a hill/ravine that we must descend, SGD with momentum uses a physical approach by considering that we are “rolling” the parameter vector down that hill. As such, Under this approach, the parameter vector has a velocity \mathbf{v} which is used to determine its position (i.e. the update). This velocity is directly affected by the gradients that we compute:

$$\mathbf{v} = \mu \mathbf{v} - \alpha \Delta \mathbf{w}_{batch} \quad (12)$$

$$\mathbf{w} = \mathbf{w} + \mathbf{v} \quad (13)$$

Where α is the learning rate as in (11) and $\mu \in (0,1]$ is referred to as the momentum, although from a physics standpoint, it can be assimilated to friction and is generally set to 0.9.

SGD with momentum almost always boasts better convergence rates than vanilla SGD. This is because as we approach a local minimum, the updates become smaller and smaller (as the gradient tends to 0 at stationary points) so vanilla SGD

takes more time to converge to that minimum. Adding momentum on the other hand ensures that this slow-down does not occur or is minimised.

2.4.4.3 Adagrad

The next set of parameter updating techniques possess the particularity of having a per-parameter adaptive learning rate. Adapting the learning rate can be viewed as refining the search for a local minimum. Indeed, we first start with a relatively large learning rate as we want to explore the gradient space with big steps. Once a region seems promising, we would like to focus on it and reduce size of the steps as we do not want to miss a local minimum. This can be done by reducing the learning rate α when we are close to a local maximum. While we can adapt the learning rate globally for all parameters, recent research has focused on methods that adapt the learning rate per-parameter. **Adagrad**, first proposed by Duchi et al.[CITE], is one of these methods and is defined as follows: suppose we have just calculated a gradient δw for a mini batch, then the update is applied as follows:

$$c \leftarrow c + (\Delta w)^2 \quad (14)$$

$$w \leftarrow w - \frac{\alpha}{\sqrt{c} + \varepsilon} \Delta w \quad (15)$$

Here c , which has the same dimensions as the parameters, acts as a cache that stores the squared values of the previous gradients. We then divide the learning rate α by the square root of c (plus some small positive constant ε to ensure that we do not divide by zero) so that the more updates a parameter gets, the smaller its learning rate.

2.4.4.4 RMSProp

One downside of Adagrad(2.4.4.3) is that the learning rate is monotonically decreasing which does not allow for flexibility as the learning rate diminishes quickly. **RMSProp** attempts to alleviate this by introducing a decay rate parameter

γ when updating the cache c . The rest is identical to (2.4.4.3). As such the only change is that Eq. (14) becomes:

$$c = \gamma c + (1 - \gamma)(\Delta w)^2 \quad (16)$$

2.4.5 Models

The reader is referred to [9] (paragraph 3) for a review of existing deep learning methodologies for facial expression recognition "in-the-wild".

In terms of deep learning models, we use two pre-existing and pre-trained models, namely **VGG** [8] and **Inception V2** [CITE]. These were fine-tuned for specific tasks on our database (a subset of the EmotioNet database[CITE])

2.4.5.1 VGG 16

The VGG 16 network was developed by Karen Simonyan and Andrew Zisserman from Oxford University's Visual Geometry Group and was the runner-up in ILSVRC 2014. The "16" signifies that there are 16 trainable (weight) layers: 13 convolutional layers and 3 fully connected layers, see Fig. 8.

It's originality comes from the fact that it uses small kernel filters (3×3) at each convolutional layer which are convolved with every pixel of the input image (i.e. they use a stride of 1). Furthermore, the number of filters increases as we progress through the convolutional layers. The idea behind this is to get the first layers to detect abstract features such as lines and get the following layers to refine on that feature.

Finally all 5 max pool layers use a spacial extent of $p = 2$ and a stride of $s = 2$ so that each pool layer divides the width and height of the input image by 2

2.4.5.2 Inception V2

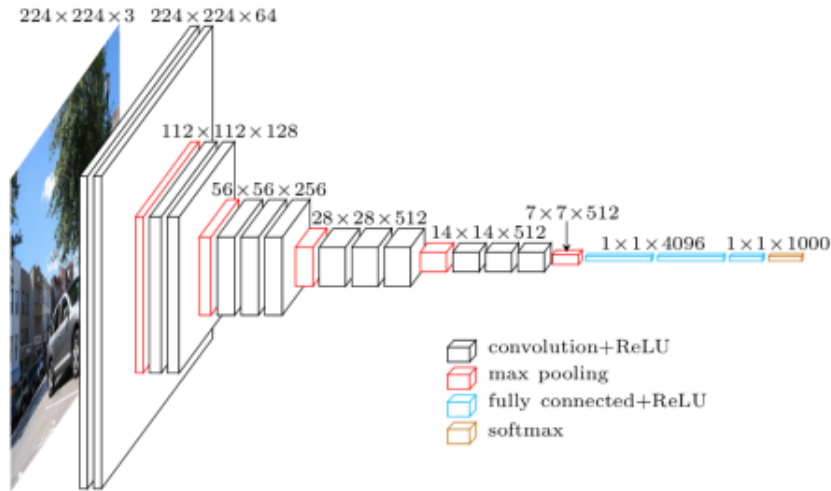


Figure 8: structure of the VGG 16 deep convolutional network

Source: <https://www.cs.toronto.edu/frossard/post/vgg16/vgg16.png>

Inception networks build upon a key insight from the Network in Network (NiN) paper[CITE, Lin et al 2014]: convolutional layers can only learn linear combination of the inputs, to enable them to learn non linear features, Lin et al replaced their linear kernel filters with multilayer perceptrons.

Researchers at Google used this insight to create the inception module which for the first version of the network (v2) is detailed in Fig. 9

Indeed, the 1×1 convolution layer is mathematically equivalent to a multi-layer perceptron and since it is immediately followed by a ReLU layer (not shown in Fig. 9), it allows the network to learn non linear features.

Furthermore, this initial 1×1 convolution layer drastically reduces the number of trainable parameters when it is used before the 3×3 or 5×5 convolutional layers. This therefore allows the Inception network to combine a max pool, a perceptron, a 3×3 and a 5×5 convolution operation into a single layer/module at no extra computational cost, effectively allowing the network to be even deeper.

With the second version of the Inception network, which we will be using, the inception module depicted in Fig. 9 differs slightly. Indeed, the authors factorise the 5×5 convolutional layer into two consecutive 3×3 convolutional layers as shown in Fig. 10.

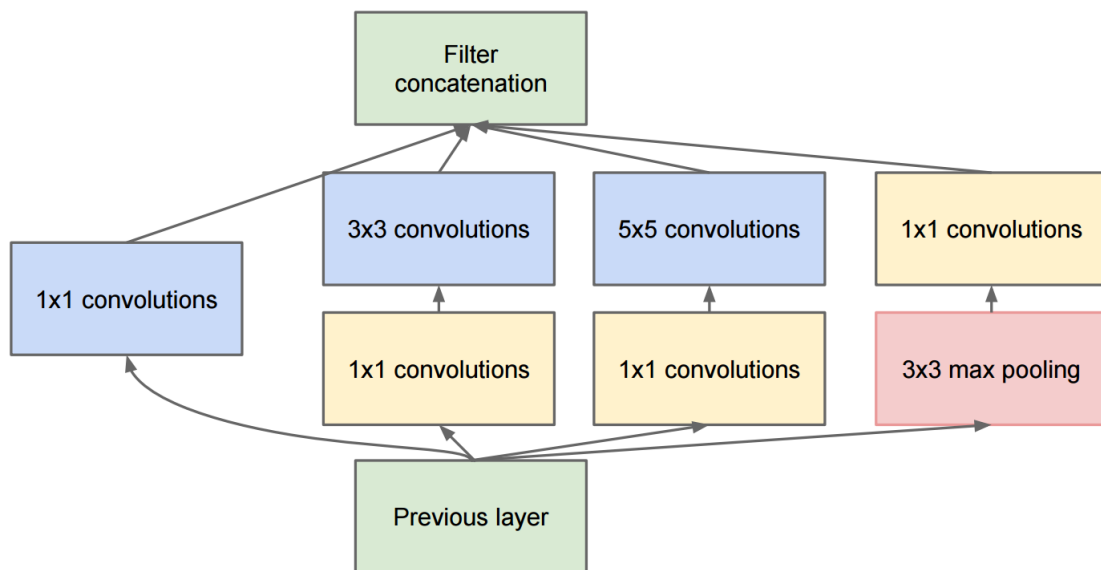


Figure 9: The inception module with the concatenation of a pool, a 1×1 , a 3×3 and a 5×5 convolution operations

Source: <https://i.stack.imgur.com/zTinD.png>

2.4.6 Evaluation Measures

Once a model has been trained, we would like to evaluate its performance on an independent and unseen test set. To do this various measures exist to provide meaningful insights about the quality of the model.

2.4.6.1 True/False Positives and Negatives

Consider a binary classification task, e.g. an action unit is either active (positive class) or inactive (negative class). Then a model's prediction for this task can fall into one four categories. If the model predicts an example to be positive and the actual value is also positive, then it is a *True Positive* (TP). If it predicts the example to be negative when in fact it actually is positive then this is a *Type II* error and the prediction is a *False Negative* (FN).

Conversely, if the example's actual label is negative and the model predicts it to be positive, then it is a *Type I* error and the prediction is a *False Positive* (FP). Finally if the model predicts the example to be negative when it is in fact negative,

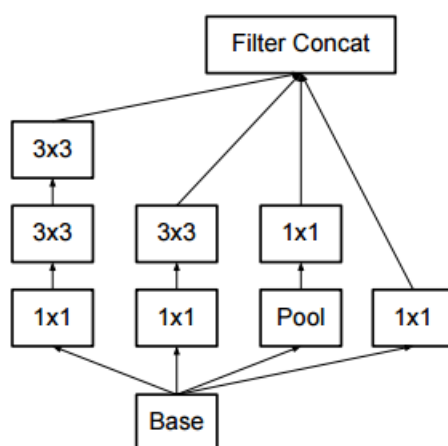


Figure 10: The inception module with the concatenation of a pool, a 1×1 , a 3×3 and a 5×5 convolution operations

Source: http://davidstutz.de/wordpress/wp-content/uploads/2017/03/inception_arch_1.png

then it is a *True Negative* (TN).

Using these simple definitions, we can now define more complex evaluation measures.

		Prediction outcome		total
		p	n	
actual value	p'	True Positive	False Negative	P'
	n'	False Positive	True Negative	N'
total		P	N	

Figure 11: The four categories a model's prediction can fall in

Source: <https://i.stack.imgur.com/00xEo.png>

2.4.6.2 Accuracy

2.4.6.3 Partial Accuracy

2.4.6.4 Recall

2.4.6.5 Precision

2.4.6.6 F1 Measure

3 EmotioNet Database

3.1 Downloading

In order to use less space, the authors of the EmotioNet database do not offer a direct download of the around 25k images. Instead, they provide an `xlsx` file containing one line per image which consists of 61 columns: the first column corresponds to the URL of the image and the next 60 columns correspond to the 60 AUs in ascending order. To download the database, one must therefore read this file line by line, get the image at the specified URL and store it in a file.

3.1.1 Reading the `xlsx` file

Python does not natively support the reading of `xlsx` files, and even though third party packages such as **`openpyxl`** by Eric Gazoni and Charlie Clark exist, we chose to convert the `xlsx` file containing the URLs and labels to a CSV file as these are easily readable in Python. One problem arose using this method, some URLs (less than 15) contained commas, meaning that the URL was split up into two or more columns which shifted the index of the AU values, thus giving an invalid label to

the associated image. To remedy to this situation, we simply deleted all the commas from the URLs.

3.1.2 Downloading the image

Once we have retrieved the URL, we can get the raw image over the Internet by using the `urllib` library. This did not come without some complications as 2,760 images were not downloaded due to the errors listed below:

- HTTP 404 (Not Found) error: 1,726 URLs
- HTTP 400, 401, 403, 410 errors: 205 URLs
- HTTP 500, 502, 503, 504 errors: 102 URLs
- HTTP 301, 302 errors: 20 URLs
- Name or service not known: 500 URLs
- Connection timed out: 109 URLs
- Connection reset by peer: 17 URLs
- Connection refused: 16 URLs
- No route to host: 10 URLs
- Certificate verify failed: 14 URLs

The *Name or service not known* errors can be caused by the lack of an internet connection which can occur when downloading via Wi-Fi. Because of this, some valid URLs will not be accessed to download an image. To solve this problem, we keep track of which images remain to be downloaded by taking all the URLs and removing those that were already downloaded as well as those that led to an error and launch the download script again. To keep track of the erroneous URLs, each time an URL leads to an error, it is stored along with its error in a text file and to construct the list of already downloaded images, we just list the files present in the specified download directory.

3.1.3 Storing the image

Storing the fetched raw image implies creating a file and more importantly naming it in such a way that using this name, we can retrieve the associated label (AU values) from the `xlsx` file. We initially used an URL parser to extract the query component of the URL which should consist of the name of the image (e.g. `image.jpeg`). However, it turns out that this query component can also have a parameter component (e.g. `?size=200x200` in `HTTP://www.foo.com/bar/image.jpeg?size=200x200`). So converting the URL into a file name based solely on the query component of the URL meant that two different URLs/images could lead to the exact same file name when we require this `url_to_filename` function to be a bijection in order to be able to retrieve the associated label from the `xlsx` file.

We therefore abandoned this method and just used the URL with a couple of preprocessing steps as file name. The preprocessing steps consisted in removing full stops from the URL and replacing forward slashes with underscores as they would otherwise be understood as directories by the file system. Additionally, we determine the type of the raw image (e.g. `png`, `jpeg`, `gif` ...) and append it to the file name. Finally, file names cannot be longer than 250 chars on most systems so we truncate the beginning of URLs that violate this limit.

3.2 Converting to TFRecords

TFRecords is the standard recommended file format to store data that will be used by a TensorFlow network or Graph. It is based on Google's Protocol Buffers which are language and platform agnostic object serialisation mechanisms.

Converting a data set to a set of TFRecord files is no easy task, but thankfully TensorFlow provides a script, `build_image_data.py`, that does just that and only needed a few modifications to adapt it to the nature of our data set. Indeed, this script provided by TensorFlow was used to convert the images of the CIFAR-10[CITE] database and relied on a special organisation of the data to assign labels: one eponymous subdirectory should be created for each image category (label) and all the images of that category should be placed into that subdirectory (e.g. and image of a cat would be in the `/dataset/cat/` subdirectory). However this mechanism

does not apply in our case since it assumes that the categories are mutually exclusive (one image cannot be categorised as a cat and a dog at the same time) to partition the data set into subdirectories, whereas in our case, an image can have multiple active AUs at once.

So we had to change the way the script assigned labels to images by constructing a dictionary of URLs to labels using the `xlsx` data file and then using the file name of the image to look up its label in this dictionary. We also had to change the way the label was stored in the `TFRecord` as it now consist of an array of `int64` rather than a single `int64`. Finally, the script converts any PNG image to JPEG for consistency, however, it does not handle other image types such as GIF so we added the necessary logic to convert GIFs to JPEG images.

The script then takes care of multi-threading for efficiency and outputs a training and a validation `TFRecord` file, both split into a number of user-specified shards.

3.3 Extending TF-Slim datasets

Having downloaded the database and converted it to `TFRecord` format, we now extend the TF-Slim datasets to incorporate ours. This has many advantages as TF-Slim then takes care of creating a data provider for our models and more importantly, it allows us to easily reuse TF-Slim's generic training and evaluation scripts.

In order to do this, we created a new file in `/slim/datasets/` called `images.py` which returns an instance of a TF-Slim Dataset class, customised for our dataset. We then add this instance to the `dataset_factory` file's dictionary. Our dataset can now be accessed using the `dataset_factory.get_dataset()` method which only requires the name of the dataset, the path to the actual data and the name of the split to use (one of 'train' or 'validation'). This means that we can easily change data sets, should we add some in the future and makes our training and evaluating system more modular.

3.4 Annotating Valence and Arousal

As mentioned in section [CITE], this database was annotated by human annoators on Action Unit presence/absence only. However, we are also intereseted in valence and arousal values as it is believed that the presence or absence of AUs is correlated to the valence and arousal of the emotion that can be interpreted from a facial expression.

As such, one of the major tasks, was to first annotate the 21,500 images that were downloaded with both valence and arousal. This was done using a tool available for research purpose only and developed by J. Kossaifi, G. Tzimiropoulos, S. Todorovic and M. Pantic. This tool was designed to annotate videos, not large image data sets, so we first divided the 21,500 images into 21 videos of 1000 frames and a 22nd video of 500 frames. You then load up a video and start annotating the frames for valence. Once all the frames of the video are annotated with valence, we annotate them with arousal and move on to the next video.

4 Action Unit Prediction

The first deep learning task consisted in fine-tuning a model for AU classification on the EmotioNet data set that was downloaded in the previous section. We used two different pre-trained models, VGG 16 and Inception V2, to this end.

4.1 Adapting training scripts

4.1.1 Duplicating the Logits

The `tensorflow/models/slim` source code provides a generic training script to train or fine-tune VGG and Inception models on different data sets. This training script offers considerable flexibility as virtually any training parameter can be specified using and exhaustive set of flags.

However, this training script expects 1-dimensional labels (e.g. 'Car') when

our data has 2-dimensional labels. This caused problems when trying to compute the loss function as there was a mismatch between the logits (the outputs of the network) and the labels. Indeed, the script performs one-hot encoding on the labels which adds a dimension and transforms each label into a 60×60 matrix whereas the logits are a vector of 1×60 . To solve this problem, we simply duplicated each logit by the number of classes (60) so as to obtain a 60×60 matrix with identical rows (equal to the original logit).

This works from a mathematical view point as we are using the softmax cross entropy loss function (see (2.4.3.2)) and so each entry in this new logit matrix will be multiplied by its corresponding entry in the one-hot label matrix. But since it is one-hot, only one entry per row is 1 which means that the one-hot label matrix effectively selects the appropriate logit entry and disregards the others by multiplying them by zero.

4.1.2 Cleaning the Labels

In the labels provided by the data set, each action unit can take on 3 values: 0 for inactive, 1 for active and 999 for undefined. However, we would like them to only take on 2 discrete values: 0 and 1. As such, it was necessary to preprocess the labels before feeding them to the loss function in order to replace every 999 by a 0. We will later see that this does not lead to desirable results.

4.1.3 Changing the final activation function

Originally, the training script used `softmax_cross_entropy_with_logits` as its loss functions. The unscaled logits were therefore passed through a softmax (2.4.2.2) function before computing their cross entropy. This led to some very poor results with the training loss growing to numbers greater than $10e^6$. This is because the softmax function calculates the probability of the training example/image to be one of the classes, therefore assuming that these classes are mutually exclusive so that that the image can not belong to two classes at the same time. However, this is not the case for our data set as images can have multiple AUs/classes active at the same moment. As such, the probabilities calculated by the softmax function will be

low (as there is no clear “winner” class) which gives large values when we take the negative log of these probabilities in the cross entropy.

Therefore, the `softmax_cross_entropy_with_logits` loss function is not adapted to our situation. Instead, we must treat each action unit as an individual binary (is it activated or not, i.e. 1 or 0 respectively) classification task. This can easily be done by replacing the softmax activation function by a sigmoid activation function (2.4.2.1). Indeed, the sigmoid function does not normalise each value in the logit array with respect to the others, thus treating each class as independent of one another. As such all that had to be done was to replace the current loss function with the following `sigmoid_cross_entropy_with_logits`. This also mean that it was unnecessary to duplicate the logits before passing them to the loss function as no one-hot encoding of the labels is necessary.

4.2 Prediction of 60 Action Units

We started the action unit prediction task with the whole set of action units. That is, for each training image, we learn to predict if each of the 60 AUs are either active or inactive. Note that as discussed in (4.1.2) we marked the AUs labelled as occluded as inactive. Though this is should not be done, it was impossible to simply remove any example whose label contained a '999' as we would be left with an empty data set since each label contained at least one AU marked as '999'.

4.2.1 Fine-tuning

The training consisted in fine-tuning the VGG 16 and Inception V2 (see 2.4.5) on a data set split of 15,000 examples. For the VGG 16 models, this meant training the weights of the final three convolutional layers whilst freezing the remaining convolutional layers whereas for Inception V2, this meant training the final logits layer whilst freezing the rest of the network.

Both networks were trained using the RMSprop optimiser (2.4.4.4) using a decay rate of $\gamma = 0.9$ and a momentum parameter of 0.9 as well. We also used an initial learning rate of 0.01 in both cases, with an exponential decay using a

decay factor of 0.94 and an interval of 2 epochs between two decays of the learning rate. The batch size was set to 32 and finally a weight decay of 0.00004 for the L2 regularisation term.

We carried out the fine-tuning for 20,000 steps which amounts to about 43 epochs.

4.2.2 Evaluation

Evaluation was carried out on a separate and independent data set split of 6400 images.

For each model, we report the total accuracy, the partial accuracy, the recall, the precision, and the F1 measure.

5 Valence and Arousal Regression

5.1 VGG 16

5.2 ResNet

6 Future Work

References

- [1] C. Fabian Benitez-Quiroz, Ramprakash Srinivasan, and Aleix M. Martinez. Emotionet: An accurate, real-time algorithm for the automatic annotation of a million facial expressions in the wild. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 5562–5570, 2016. pages 7, 8
- [2] Paul Ekman and Wallace V. Friesen. Constants across cultures in the face and emotion. *Journal of personality and social psychology*, 17(2):124, 1971. pages 4, 5
- [3] Paul Ekman, Wallace V. Friesen, and Joseph C. Hager. Facial action coding system (facs). *A technique for the measurement of facial action*. Consulting, Palo Alto, 22, 1978. pages 4, 5
- [4] B. Fasel and Juergen Luetttin. Automatic facial expression analysis: a survey. *Pattern Recognition*, 36(1):259–275, 1 2003. pages 4
- [5] Daniel McDuff, Rana Kaliouby, Thibaud Senechal, May Amr, Jeffrey Cohn, and Rosalind Picard. Affectiva-mit facial expression dataset (am-fed): Naturalistic and spontaneous facial expressions collected. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, pages 881–888, 2013. pages 7
- [6] Maja Pantic and Leon J. M. Rothkrantz. Automatic analysis of facial expressions: The state of the art. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 22(12):1424–1445, 2000. pages 7
- [7] RL Schiefelbusch. *Handbook of methods in nonverbal behavior research*. Klaus R. Scherer & Paul Ekman (Eds.). Cambridge University Press, 1982., volume 4. Cambridge Univ Press, 1982. pages 5
- [8] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *CoRR*, abs/1409.1556, 2014. pages 20
- [9] Stefanos Zafeiriou, Athanasios Papaioannou, Irene Kotsia, Mihalis A. Nicolaou, Guoying Zhao, E. Antonakos, P. Snape, G. Trigeorgis, and S. Zafeiriou. Facial affect in-the-wild: A survey and a new database. In *International Conference on Computer Vision*. pages 4, 7, 20