Imperial College London

FINAL REPORT

IMPERIAL COLLEGE LONDON

DEPARTMENT OF COMPUTING

Deep Learning For Facial Expression Analysis

Author:

Tom Bartissol (CID: 00824562)

Date: June 21, 2017

Abstract

Humans can communicate through multiple channels including speech and facial expressions. Enabling machines to understand these channels would unlock a multitude of opportunities for Human Computer Interaction and while progress has been made on the speech side, facial expressions remain more complex for a machine to understand.

However, with the advent of Deep Learning models in recent years, this goal might be within reach. We use two such models, Inception V2 and VGG 16, to classify Action Units with a maximal accuracy of 88.9% on a subset of EmotioNet and to predict Valence and Arousal of facial expressions with a best root mean squared error of 3.71.

Acknowledgements

I would like to thank the following people for their contributions to this project:

- my project supervisor, Stefanos Zafeiriou
- PhD student Dimitris Kollias for taking the time to patiently answer my questions
- my friends and family for their continuous and unconditional support

Contents

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* [3] (see Table 2), or better yet, on two continuous emotion dimensions, 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) [4] 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 whether these AUs are active or not 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. We are therefore interested in images of facial expressions. 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 [16], we are interested in the so called "*in-the-wild*" images, which are taken 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 VGG 16 and Inception V2. Each model will be fine-tuned to solve problems (i) and (ii) on data taken in-the-wild, and more precisely on the EmotioNet [1] database Finally we will evaluate these models on an independent subset of the same database.

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 [13], the most popular and used coding system is the FACS [4]. 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, see Fig. 2.

2.1.2 Interpreting emotions

2.1.2.1 Discrete case

Emotions can roughly be broken down into six *basic emotions* [3], seven if we count the neutral emotion. We can then interpret a facial expression into one of these

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

seven categories. Note that this is only an interpretation as emotions are expressed through 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 aversivenes of an emotion, that is respectfully how positive or negative the emotion is.
- **Arousal**: characterises the intensity of the emotion, as such high arousal would indicate a state of increased activity and alertness of mind and body (e.g. fear or excitement).

We can therefore represent emotions in a two dimensional coordinate system using their valence and arousal values, see Figure 3

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 [16], the interested reader is directed towards [10, 12] for a thorough overview of these early techniques.

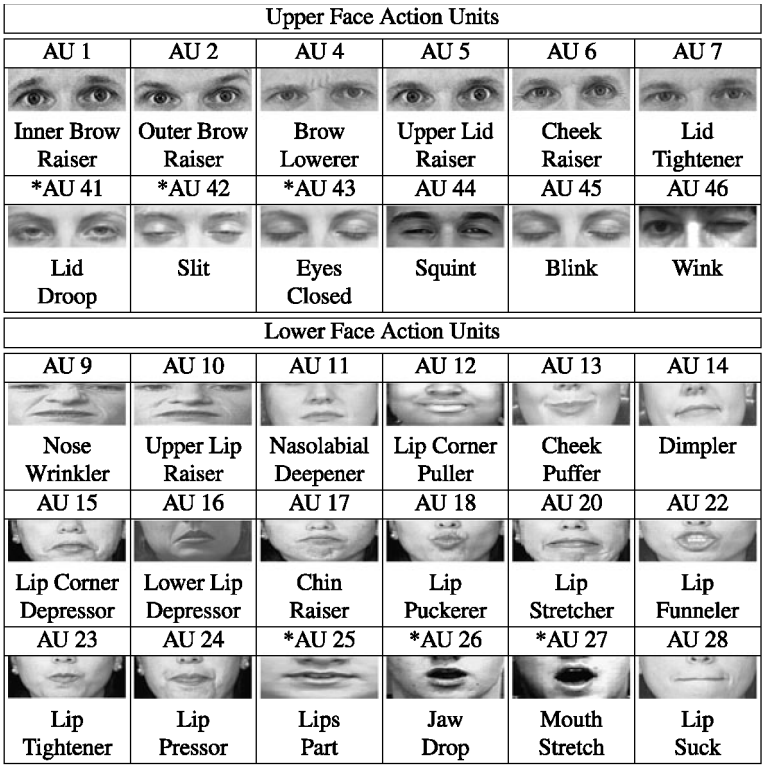


Figure 1: Facial Action Units

2.3 Database - EmotioNet

We direct the reader towards [16] 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 EmotioNet [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 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. 4

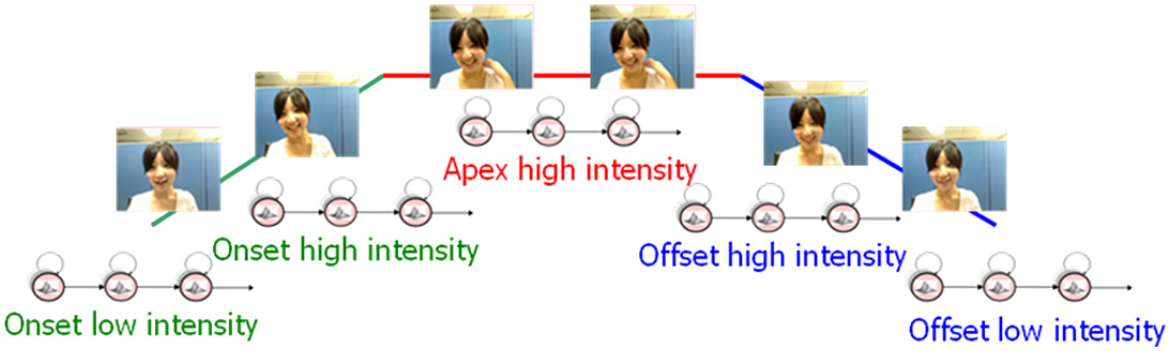


Figure 2: The neutral-onset-apex-offset-neutral cycle of an action unit activation

Source: http://proj.ncku.edu.tw/research/articles/e/20150508/images/150427015117V6MKPn.jpg

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

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 layers. Let a_j^i be the output, or activation, of the j^{th} neuron of the i^{th} layer for $i = 1 \ldots n$ 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:

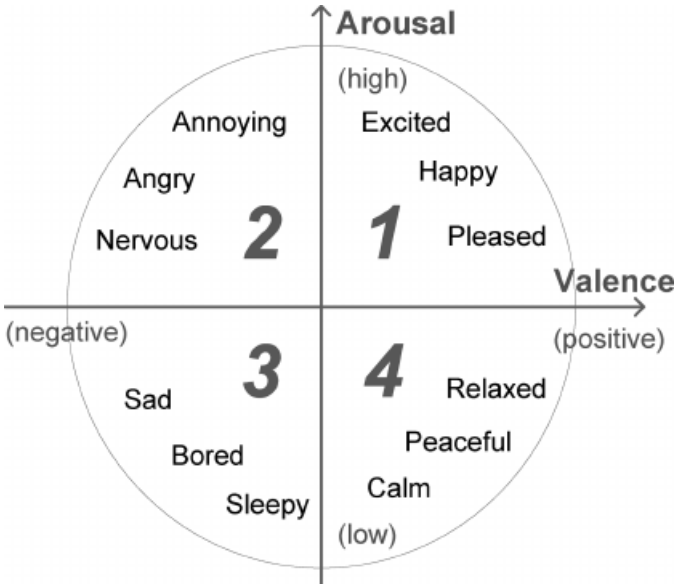


Figure 3: Plotting emotions according to their valence and arousal

Source: https://www.researchgate.net/profile/Yi-hsuan_Yang/publication/254004106

$$a_j^i = f\left(z_j^i\right) \tag{1}$$

$$z_j^i = s_j^i + b_j^i \tag{2}$$

Where s_j^i is the some weighted sum of all or part of the activations a_j^{i-1} for $j = 1 \ldots m_{i-1}$ (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 $f(.)$ 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 $\in \mathcal{R}^{nxm}$ where n is the image height and m the image width, either in greyscale ($\in \mathcal{R}^{1xnxm}$) or in RGB colours $\in \mathcal{R}^{3xnxm}$ (where 3 is the depth of the image, i.e. the number of channels) and processes it to

Figure 4: 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'

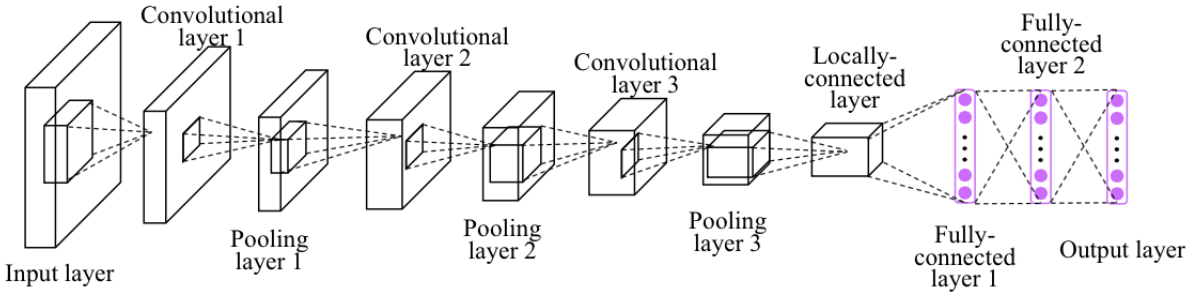


Figure 5: example of a DCNN architecture with a succession of convolutional + max pooling layers followed by a bloc of fully connected layers

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

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 were 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 7x7xD where the first and the second 7 correspond to the height and the width of the filter respectively and D corresponds to the depth (e.g. 1 for greyscale 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 $I \in \mathcal{R}^{WxH}$ and a filter $F \in \mathcal{R}^{kxk}$ 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 $C(\cdot,\cdot)$ executed at pixel (a,b) of the image is

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

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. 6. Intuitively, each kernel filter learns to detect specific features in the input space such as as 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 and 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

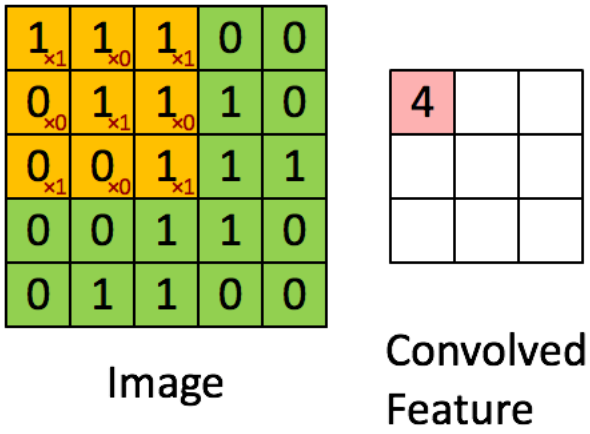


Figure 6: Example of a convolution operation at a single spatial location. The large yellow square represents the kernel filter. Note that no padding has been applied here which is why the convolved feature (i.e. the response map) is smaller.

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

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 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 7.

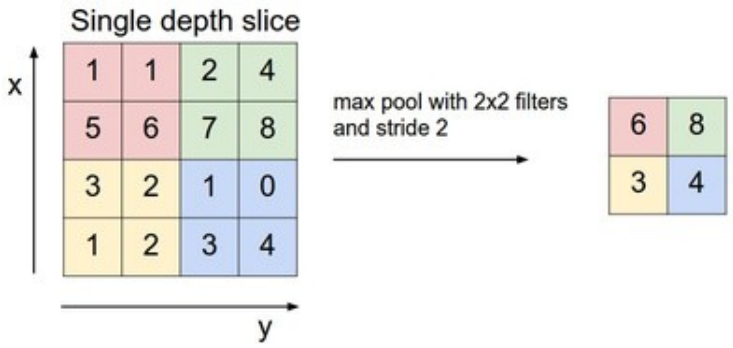


Figure 7: example of a single max pooling operation with a 2×2 pooling filter and a stride of 2

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 DCNNs, after the block of convolutional + max pool layers.

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 are simply calculated via matrix multiplication between the inputs and their associated weights and adding a bias term before passing this sum to an activation function (see Section 2.4.2).

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

[14] 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 [15] 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. 8. This prevents groups of neurons from *co-adapting* too much and improves the flexibility of the network which *decreases overfitting*.

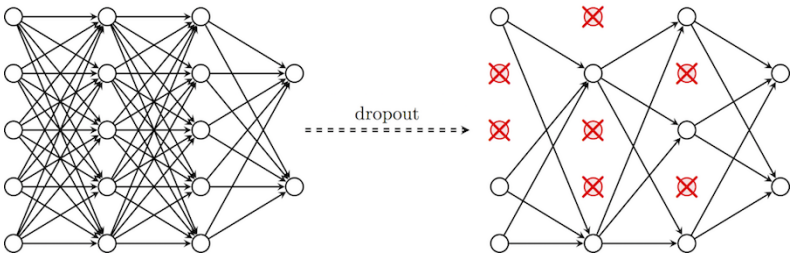


Figure 8: example dropout layers: the crossed neurons are blocked from sending their inputs to the next layer

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

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 functions is to provide non-linearity to the network. Intuitively, this enables deep networks to better fulfil 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:

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

It was a popular activation function until the arrival of the ReLU (2.4.2.3) which dealt better with 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 indeed problematic as when performing backpropagation, the updates will not change the weights significantly which considerably slows down 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 mostly 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 $\boldsymbol{z}^i$ be the vector of raw activations of this layer as defined in (2), then

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

Where m_i denotes the number of neurons in layer i.

From a probability theory point of view, 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 [11] for a scalar input x is defined as follows:

$$f(x) = \max(0, x) \tag{6}$$

Variations have recently emerged such as the Leaky ReLU [8] which is defined as:

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

There are two main advantages to using ReLUs which make them the most popular activation functions today. The first is that they significantly reduce 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 and be less affected by noisy dependencies.

2.4.3 Loss functions

Loss functions are at the core of deep neural networks as they define the objective that we are trying to achieve. It is therefore crucial to select the appropriate loss function depending on the task for which the network will be trained.

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 $(\boldsymbol{x}_i, y_i)_{i=1}^{N}$ with N examples, it is defined as:

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

Where $\hat{y}_i$ is the network's prediction using input $\boldsymbol{x}_i$.

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

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

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,

$\Delta \boldsymbol{w}$, is computed using the backpropagation algorithm and the most simple way to apply it to update the parameters $\boldsymbol{w}$ is as follows:

$$\boldsymbol{w} \leftarrow \boldsymbol{w} - \alpha \Delta \boldsymbol{w} \tag{10}$$

where α, the learning rate, controls the magnitude of the updates. In this setting, $\Delta \boldsymbol{w}$ 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 $\Delta \boldsymbol{w}_{batch}$ w.r.t. the parameters after a forward pass of a small batch $(\boldsymbol{x}_{batch}, \boldsymbol{y}_{batch})$ of training examples, usually 128 or 256, instead of the entire dataset. It then updates the parameters as in Eq. (10):

$$\boldsymbol{w} \leftarrow \boldsymbol{w} - \alpha \Delta \boldsymbol{w}_{batch} \tag{11}$$

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 and 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 v which is used to determine its position (i.e. the update). This velocity is directly affected by the gradients that we compute:

$$\boldsymbol{v} = \mu \boldsymbol{v} - \alpha \Delta \boldsymbol{w}_{batch} \tag{12}$$

$$\boldsymbol{w} = \boldsymbol{w} + \boldsymbol{v} \tag{13}$$

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. [2], is one of these methods and is defined as follows: suppose we have just calculated a gradient $\Delta \boldsymbol{w}$ for a mini batch, then the update is applied as follows:

$$\boldsymbol{c} \leftarrow \boldsymbol{c} + (\Delta \boldsymbol{w})^2 \tag{14}$$

$$w \leftarrow w - \frac{\alpha}{\sqrt{c} + \varepsilon} \Delta w \tag{15}$$

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:

$$c = \gamma c + (1 - \gamma)(\Delta w)^2 \tag{16}$$

2.4.5 Models

The reader is referred to [16] (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** [14] and **Inception V2** [5]. These were fine-tuned for specific tasks on our database (a subset of the EmotioNet database [1])

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. 9.

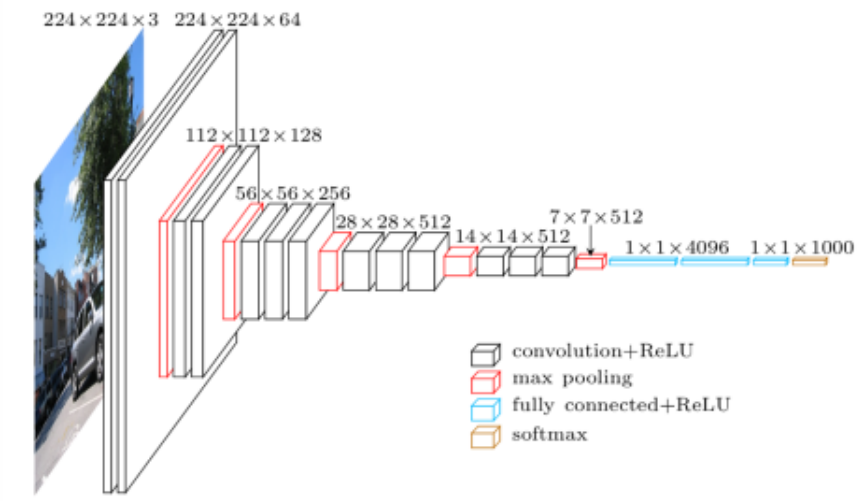


Figure 9: structure of the VGG 16 deep convolutional network

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

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

Inception networks build upon a key insight from the Network in Network (NiN) paper [7], 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. 10

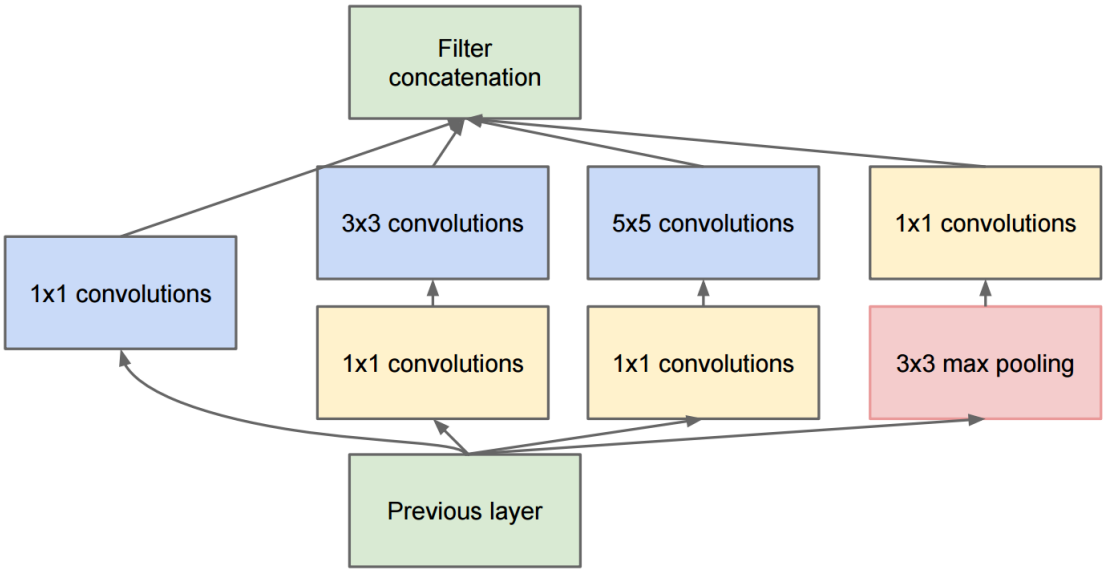


Figure 10: The inception module with the concatenation of a pool, a 1×1, a 3×3 and a 5×5 convolution operations

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

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. 10), 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. 10 differs slightly. Indeed, the authors factorised the 5×5 convolutional layer into two consecutive 3×3 convolutional layers as shown in Fig. 11.

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 mean-

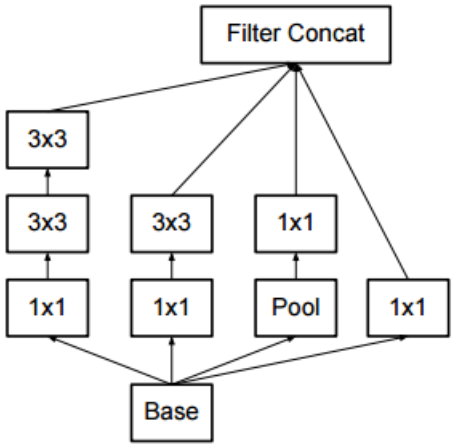


Figure 11: The inception module with the concatenation of a pool, a 1×1, a 3×3 and a 5×5 convolution operations

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

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

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

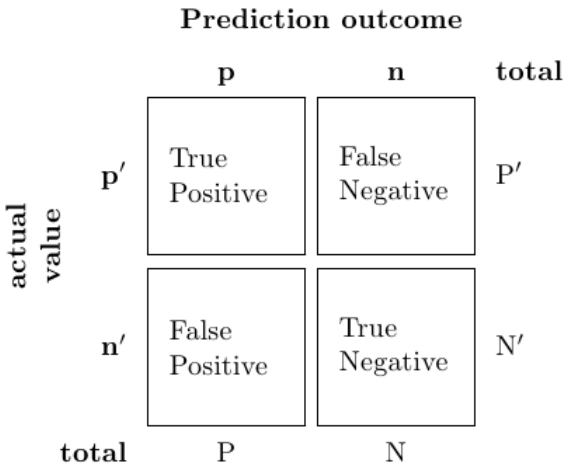


Figure 12: The four categories a model's prediction can fall in

Source: https://i.stack.imgur.com/0OxEo.png

2.4.6.2 Accuracy

Accuracy is the simplest and most intuitive measure, it measures the proportion of examples that were correctly classified as positive or negative and is defined as:

$$Acc = \frac{TP + TN}{TP + TN + FP + FN} \tag{17}$$

2.4.6.3 Partial Accuracy

Suppose the label $\boldsymbol{y}$ is as in the EmotioNet data set, i.e. an array composed of 60 entries, each corresponding to the AU of its index, which are either 1 if the AU is active or 0 otherwise. Now suppose we also have a prediction $\hat{bmy}$ of same size and also composed of zeros and ones. With normal accuracy (2.4.6.2), if as little as one $\hat{\boldsymbol{y}}_i$ differs from
bmy_i, for $i = 1 \dots 60$, then the prediction is assumed to be incorrect as a whole, as it does not match the label exactly, entry for entry. This is not desirable when the classes are not mutually exclusive as the other entries (e.g. the other AUs) are

correctly classified. We are therefore interested in the proportion P_i of classes (e.g. AUs) that are correctly predicted within a single label $\hat{\boldsymbol{y}}^i$:

$$P_i = \frac{1}{m} \sum_{k=1}^{m} I(\hat{\boldsymbol{y}}_k^i, \boldsymbol{y}_{k^i}) \tag{18}$$

Where m is the number of classes and $I(a,b) = 1$ iff $a = b$ and 0 otherwise. We then take the average of the P_i over $i = 1 \dots N$ to output the average partial accuracy of the model over a test set of N examples.

2.4.6.4 Recall

Recall measures the proportion of *actually* positive examples that where *predicted* as being positive and is defined as:

$$Recall = \frac{TP}{TP + FN} \tag{19}$$

Note that one can achieve a Recall of 1 (the highest value possible) by predicting every example as positive.

2.4.6.5 Precision

Precision measures the proportion of examples that *actually* are positive within the set of examples that were *predicted* to be positive. It is defined as:

$$Precision = \frac{TP}{TP + FP} \tag{20}$$

2.4.6.6 F1 Measure

The F1 measure is the harmonic mean of the recall (19) and the precision (20) and is defined as:

$$F1 = 2 \times \frac{1}{\frac{1}{recall} + \frac{1}{precision}} = 2 \times \frac{precision \times recall}{precision + recall} \tag{21}$$

Intuitively, it measures the balance between precision and recall, with an F1 score of 1 signifying perfect balance (i.e. both precision and recall are 1).

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 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 ray 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 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 (2.3), this database was annotated by human annotators on Action Unit presence/absence only. However, we are also interested in valence and arousal values as it is strongly 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, is to first annotate the 21,500 images that were downloaded with both valence and arousal. This is1 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 0 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 lead 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 $5,000$ steps which amounts to about 11 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, the F1 measure and the AUC (see section 2.4.6) as shown in Table 3.

Measure \ Model	VGG 16	Inception V2
Total Accuracy	0.97710675	0.97671616
Partial Accuracy	0.97710687	0.97671622
Recall	0	0.12441715
Precision	0	0.46832192
F1 Score	0	0.19660347
AUC	0.5	0.56055355

Table 3: Evaluation measures for AU prediction/classification on the full set of AUs (AUs 1 to 60)

The first figure that strikes us when looking at the results listed in Table 3 is the abnormally high total and partial accuracy for both networks, as well as how close these accuracies are. Such high results, coupled with low recall and precision should alarm us and after a quick analysis of the labels provided with this data set, one can understand why we obtain such high figures.

Indeed, out of the 60 action units available, only 11 of them (AUs 1,2, 4, 5, 6, 9, 12, 17, 20, 25 and 26) were truly annotated. These AUs correspond exactly to those that were annotated in the DISFA [9] database with the only difference being that this database was annotated for an additional AU, namely AU 15. The remaining 49 AUs were marked as '999' (as being occluded) for almost every image of the test and training set. Furthermore, since we "clean" the labels as described in (4.1.2), these action units are marked as inactive just before training. The model therefore just has to predict a value of 0 for these AUs, which it can do very accurately since these AUs are almost always 0 after cleaning.

This lack of variance in the labels can explain the high accuracy displayed by both VGG 16 and Inception V2 as both networks do not have to learn much. Moreover, since most of the entries (49 out of 60) of each label will be 0 (after cleaning), there is a heavy bias towards that value which means that the models are more likely to predict a 0 instead of a 1. As such, the low recall and precision (and thus the low F1 score) result from the fact that 0s are considered as negatives and 1s as positives. Indeed, this suggests that there are many true negatives/false negatives, since many 0s/1s are correctly/incorrectly predicted as 0s and also that

Measure \ Model	VGG 16	Inception V2
True Positives	0	1,094
False Positives	0	1,240
True Negatives	375,209	373,967
False Negatives	8,791	7,699

Table 4: Reporting the number of True/False Positives/Negatives for VGG 16 and Inception V2 trained to predict action units for the full set of action units

there are very few true positives/false positives as almost no AUs are predicted to be 1 (active). This is confirmed by Table 4.

We do note that the Inception V2 model copes better than the VGG 16 model with this lack of variance and imbalance of "classes" within the labels as it has around 1,000 true positives. However, this figure is insignificant when compared to the number of true negatives which we have artificially created by cleaning the labels.

4.2.3 Conclusion

From this initial approach to fine-tuning a deep learning model to predict if an action unit is active or not on the full set of 60 action units, we conclude that we cannot use this full set of AUs. Indeed, too many of them are marked as being occluded which created a significant class imbalance and invariance once we have "cleaned" the labels. As such, it would be better to restrict the number of AUs we would like to predict so as to limit the number of AUs marked as occluded.

4.3 Predicting 11 Action Units

Following the previous conclusion, we limit ourselves to the following 11 action units:

1. AU 1: Inner Brow Raiser

2. AU 2: Outer Brow Raiser
3. AU 4: Brow Lowerer
4. AU 5: Upper Lid Raiser
5. AU 6: Cheek Raiser
6. AU 9: Lid Tightener
7. AU 12: Lip Corner Puller
8. AU 17: Chin Raiser
9. AU 20: Lip Stretcher
10. AU 25: Lips Part
11. AU 26: Jaw Drop

Coincidently, these 11 AUs correspond to those that have been annotated in the DISFA database (minus AU 15). They were chosen for the sparsity of 999/occluded entries in their labels. Indeed, having eliminated all the other action units, we then eliminate any image that has one of the AUs listed above marked as 999. After carrying out this cleansing, we are left with a new dataset, which we will call EmotioNet-11AU, composed of 10,282 training and 4,424 test images.

4.3.1 Fine-tuning

We can now conduct a proper fine-tuning of the VGG 16 and Inception V2 models. As in the previous section, we use the RMSprop (2.4.4.4) optimiser with a decay rate of $\gamma = 0.9$ and a momentum of 0.9. The learning rate also has the same exponential decay with a decay factor of 0.94 and the same interval of 2 epochs between 2 successive decays. Finally, the weight decay for the L2 regularisation term is the same at 0.00004.

However, perform multiple trainings whilst first varying the batch size and then the initial learning rate so as to select the best hyper-parameter.

4.3.1.1 Batch Size Selection

We first wish to select an appropriate batch size so that the training has good convergence. To do so, we train the Inception V2 model for 20,000 steps, using the following batch sizes: 32, 64, 128, 256 and record the evolution of the training loss as we progress through the number of steps.

The results are presented in Fig. 13 and show that using a small batch size is not efficient as the training loss actually increases during the first 2,500 steps before declining in a linear fashion. As we increase the size if the batch size, the training loss decreases more for a given number of steps. This trend culminates with the maximum batch size we have tested, 256, as the training loss decreases exponentially up to step 4,000 where it hits a plateau of about 0.275 which batch size 128 reaches at step 8,000 and batch size 64 at step 16,000. We do note however that, since 256 is twice as much as 128 and 4 times 64, and the corresponding number of steps follow this proportion, using a batch size of 64, 128 or 256 will require a similar number of epochs to reach the plateau training. This is not true for a batch size of 32.

We therefore chose a batch size of 128 as a compromise between the number of steps required to converge and the time (in seconds) each of these steps take.

Concerning the VGG 16 model, we used a batch size of 32 as anything above that caused our server to run out of GPU memory due to the large number of parameter that this model has.

4.3.1.2 Initial Learning Rate Selection

Having picked a batch size of 128, we now focus on selecting an effective initial learning rate. To do this we train 3 Inception V2 models with initial learning rates of 0.1, 0.01 and 0.001 respectively for 3,500 steps. Fig. 14 shows the training loss against the number of steps for these three learning rates. Clearly using an initial learning rate of 0.1 leads to an unwanted increase of the training loss as we progress though the steps and so should be avoided. Using an initial learning rate of

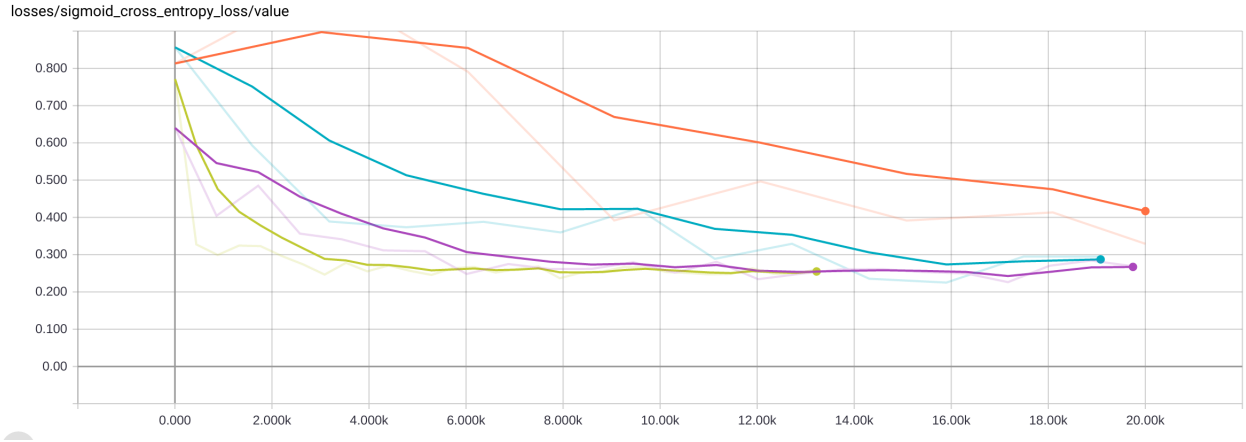


Figure 13: Training loss for different batch sizes: 32=orange, 64=cyan, 128=magenta, 256=green

0.1 or 0.001 leads to the similar type of linear decrease of the training loss, with the learning rate 0.001 boasting the best results. We therefore select $\alpha = 0.001$ as our initial learning rate for the Inception V2 model.

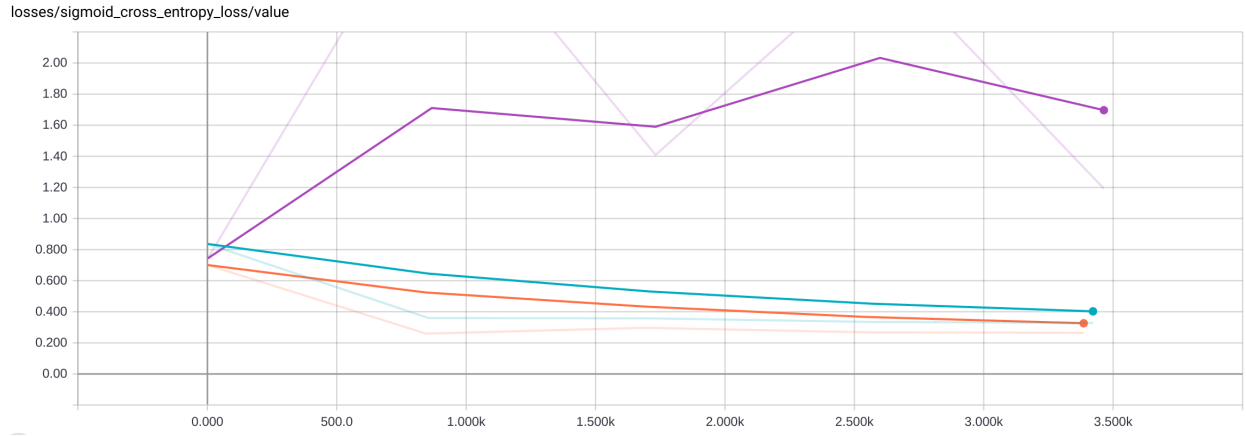


Figure 14: Training loss using Inception V2 model with the following different initial learning rates: 0.1=purple, 0.01=cyan, 0.001=orange

For the VGG 16 model, having selected a batch size of 32, we now perform the same experiment as above by training three VGG 16 model with 0.1, 0.01, 0.001 as initial learning rates respectively. The training loss of these three models is plotted in Figures 15a, 15b and 15c respectively. From these, we clearly see that the smallest initial learning rate, $\alpha = 0.001$ should be picked as the other two result either bad convergence for $\alpha = 0.01$ or extremely high training loss for $\alpha = 0.1$. Therefore, as with the Inception V2 model, we select $\alpha = 0.001$ as our initial learning rate.

Measure \ Model	VGG 16	Inception V2
Total Accuracy	0.863939	0.889071
Partial Accuracy	0.863939	0.889071
Recall	0.337218	0.434231
Precision	0.514444	0.649738
F1 Score	0.407391	0.520562
AUC	0.642984	0.698270

Table 5: Evaluation measures for AU prediction/classification on the restricted set 11 AUs

4.3.2 Evaluation

Now that we have selected some hyper parameters based on empirical results, we proceed to evaluating the Inception V2 and VGG 16 models with these parameters. Again, as with section (4.2.2), we report the total accuracy, the partial accuracy, the recall, the precision, the F1 score and the AUC in Table 5. Clearly the results are much more realistic than in section (4.2.2). We also report the number of true/false negatives and positives in Table 6, from which we note that even though there still is an imbalance between the number of 0s and the number of 1s in the labels, the models are not as biased towards predicting every action unit as '0' as they were in (4.2.2). This can mostly be explained by the increased variance of the labels in this subset of the original dataset

We note that the Inception V2 model performs better than the VGG 16 model on every measure. In particular, it has a total accuracy that is 2.51 percentage points higher than that of VGG 16.

4.3.3 Conclusion

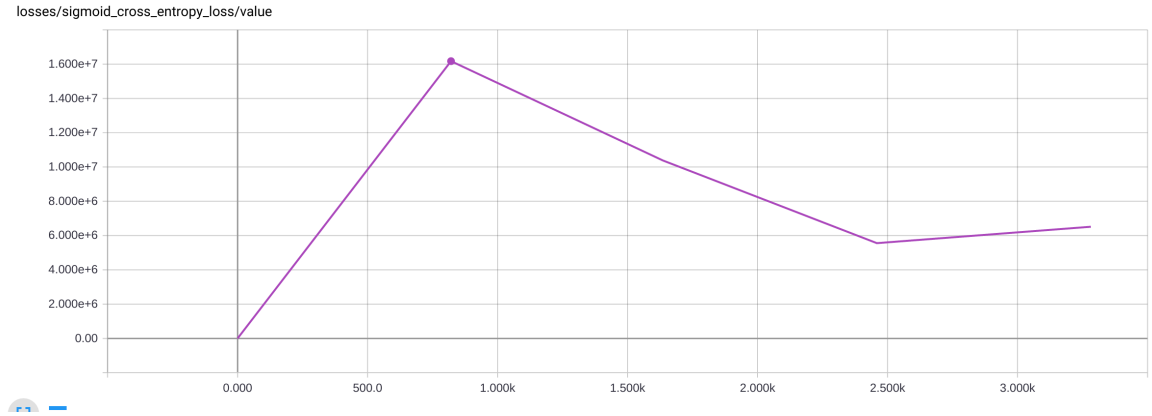
In conclusion, restricting the current task at hand to the prediction of a subset of 11 out of the 60 original action units produces more realistic models as we no longer have the problem of occluded action units.

Measure \ Model	VGG 16	Inception V2
True Positives	2,315	2,981
False Positives	2,185	1,607
True Negatives	40,450	41,028
False Negatives	4,550	3,884

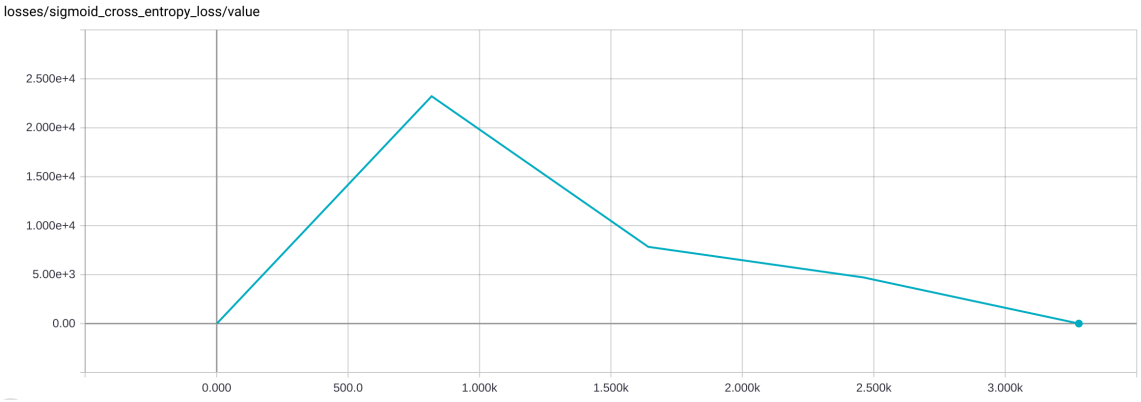
Table 6: Reporting the number of True/False Positives/Negatives for VGG 16 and Inception V2 trained to predict the restricted set of 11 action units

This was shown by training an Inception V2 with a batch size of 128 and an initial learning rate of $\alpha = 0.001$ and a VGG 16 model with a batch size of 32 and the same initial learning rate on this reduced data set, where both the batch size and the initial learning rate were selected based on empirical results to ensure better convergence during training.

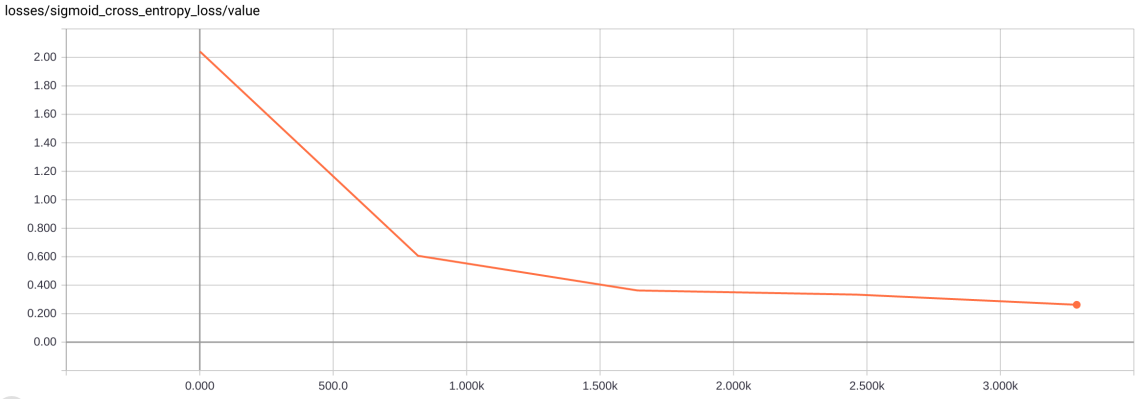
The evaluation of these two models showed realistic recall and precision measures as well as good accuracy, particularly for the Inception V2 model, which outperformed VGG 16 in every measure, which boasted a best total accuracy of 88.91%.



(a) $\alpha = 0.1$



(b) $\alpha = 0.01$



(c) $\alpha = 0.001$

Figure 15: Training loss against number of steps for the VGG model for varying initial learning rate α

5 Valence and Arousal Regression

We know switch to a new task in which the goal is to predict the valence and arousal of a facial expression. Since these two emotion dimensions are continuous, this is a regression task.

This task is of great interest as it is for this subset of the EmotioNet database. Indeed, the EmotioNet database is annotated for action unit activation or not but it is not annotated for valence and arousal.

Time constraints and other problems have led us to annotate 1,000 images from the EmotioNet database with both valence and arousal at the time of writing this report. We therefore use this small data set, divided in 800 training examples and 200 test examples, to train and evaluate an Inception V2 and a VGG 16 model.

5.1 Inception V2

We first train and evaluate Inception V2 to predict valence and arousal. To do this, we change the last layer of the network so that it only outputs two values, the first one for valence and the second for arousal. We then reuse the previous training and evaluation scripts with some slight modifications to adapt them from a classification task to a regression one, such as removing the final sigmoid activation function and changing the loss function from cross entropy (2.4.3.2) to the mean squared error.

5.1.1 Fine-tuning

As with the previous tasks, we fine-tune this model on the 800 images of the training set, still using the RMSProp (2.4.4.4) optimiser with a decay rate of $\gamma = 0.9$ and a momentum of 0.9. We also keep the same exponential decay rate of 0.94 and interval of 2 epochs between decay updates for the learning rate.

5.1.1.1 Batch Size Selection

Similarly to Section (4.3.1.1), we look for the batch size that leads to good convergence of the training loss with respect to the number of steps. To do this, we perform training with varying batch sizes and record the training loss against the number of steps as shown in Fig. 16. Interestingly, only with a small batch size of 32 do we have a monotonically decreasing training loss as we progress through the number of steps. This may be due to the high number of epochs. Indeed, since the training data set only has 800 images, training for 4,000 steps with a batch size of 128 is equivalent to training for 640 epochs. Finally, we note that the lowest training loss is achieved with a batch size of 128 and around 1,500 steps so we fix these parameters to these values.

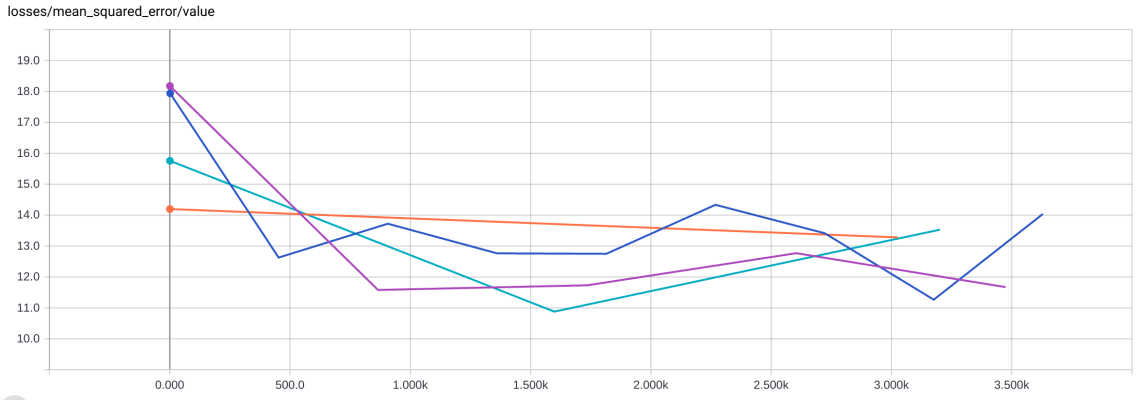


Figure 16: Training loss against number of steps for different batch sizes: 32=orange, 64=cyan, 128=magenta, 256=blue

5.1.2 Evaluation

To evaluate our model, we use the independent test set of 200 images and we report the mean squared error and the root mean squared error. For the Inception V2 network defined above, with the selected batch size of 128 and 1,500 steps, we get the following measures:

$$MSE = 15.655972 \tag{22}$$

$$RMSE = 3.9567628 \tag{23}$$

However, these are not the best results that we have found. Indeed, it turns out that using the VGG 16 model with a batch size of 32 and an initial learning rate

of 0.01 (keeping all other hyper-parameters the same as in section 5.1.1), we find the following measures:

$$MSE = 13.749833 \tag{24}$$

$$RMSE = 3.7080767 \tag{25}$$

6 Future Work

6.1 Valence and Arousal Regression

6.1.1 Larger Dataset

The small size of the data set (1,000 images) used to train the networks in sections (5.1) does not allow for an effective training of deep learning models as these thrive on large data sets.

As such, one way to cheaply increase the number of images would be to perform some data augmentation. The following operations could be applied to each image in order to multiply the size of the data set by at least a factor of 4:

1. flip image upside down or left to right
2. rotate with a uniform random angle between 0 and 360 degrees
3. random crop
4. colour perturbation

However, this does not replace the fact that the remaining images in the data set should also be annotated manually for valence and arousal. Since we are at the beginning of using machine learning to predict valence and arousal, this first manual annotation is necessary if we wish to train a model that could then replace/accelerate human annotators.

6.1.2 More Models and Full Training

It would be worth trying out more models on this data set, especially the newer versions of the Inception model (versions 3 and 4). Furthermore, building a custom model could be worthwhile, although one would probably have to pre-train it on other available data sets for better results which could be time consuming and computationally expensive.

Furthermore, we do not train the Inception V2 or VGG 16 models from scratch in this project and it would be interesting to see if performing training from scratch is more beneficial in terms of model performance. Instead of training from scratch, one could also fine-tune the whole model, not just the fully connected layers. That is, we start with pre-existing weights which are all set to be trainable in order to fine-tune them by using a small initial learning rate, say $\alpha = 0.001$.

6.2 Video Analysis

So far, we have only considered images. However, being able to continuously predict valence and arousal on video data would be extremely useful. Training a Deep Convolutional Neural Network on temporal sequences is an area of intense research as being able to interpret emotions from videos has multiple applications, including Human Computer Interaction.

6.2.1 Recurrent Neural Networks

One type model that could be used for video analysis is a Deep Convolutional Network with a recurrent neural net instead of fully connected layers after the convolutional layers. One type of recurrent network in particular is the Long Short Term Memory, or LSTM, network which is accepted as the best recurrent network for classification/regression of time series (Google uses LSTMs in their speech recognition software).

6.2.2 3D Convolutional Network

Another type of model that could be used for video analysis is a 3D convolutional neural network [6] in which convolutions are extended to the temporal dimension as well (see Fig. 17)

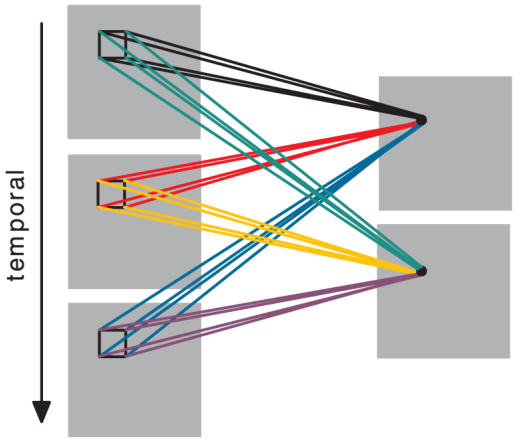


Figure 17: Example of a 3D convolution, note that connections with the same colours share the same weights

Source: [6]

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 10, 13, 26

[2] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(Jul):2121–2159, 2011. pages 25

[3] 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 10, 11

[4] 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 10, 11

[5] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015. pages 26

[6] Shuiwang Ji, Wei Xu, Ming Yang, and Kai Yu. 3d convolutional neural networks for human action recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 35(1):221–231, 2013. pages 52, 53

[7] Min Lin, Qiang Chen, and Shuicheng Yan. Network in network. *arXiv preprint arXiv:1312.4400*, 2013. pages 27

[8] Andrew L. Maas, Awni Y. Hannun, and Andrew Y. Ng. Rectifier nonlinearities improve neural network acoustic models. In *Proc. ICML*, volume 30, 2013. pages 22

[9] S. Mohammad Mavadati, Mohammad H. Mahoor, Kevin Bartlett, Philip Trinh, and Jeffrey F. Cohn. Disfa: A spontaneous facial action intensity database. *IEEE Transactions on Affective Computing*, 4(2):151–160, 2013. pages 40

[10] 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 12

[11] Vinod Nair and Geoffrey E. Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th international conference on machine learning (ICML-10)*, pages 807–814, 2010. pages 22

[12] 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 12

[13] 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 11

[14] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014. pages 20, 26

[15] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15(1):1929–1958, 2014. pages 20

[16] 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 10, 12, 13, 26