Learning Multimodal Information: Definition and Experimentation

Sunday 27th June, 2021 - 09:36

Iris Kremer
University of Luxembourg
Email: iris.kremer.001@student.uni.lu

Nicolas Guelfi
University of Luxembourg
Email: nicolas.guelfi@uni.lu

***Abstract*—This bachelor semester project seeks to explore the research field of multimodal machine learning through the production of a scientific and a technical deliverable.In the scientific deliverable, we analyse a scientific paper describing a multimodal neural network model with the goal to understand how they represent and normalize multimodal data. For the technical deliverable, we engineer a multimodal neural network on our own by joining two unimodal models together as a hands-on experience and to get familiar with some challenges of the domain. This paper presents the outcomes of the project.**

1. Introduction

Modern AI systems become increasingly powerful tools, but there remain many tasks that humans perform much better. We suppose that the multimodal perception and interpretation of the world through our five senses significantly contributes to this superior performance of humans. Involving multiple modalities in the decision making process of an AI system therefore has the potential of improving the accuracy of its predictions, and the research field of multimodal machine learning is focusing on this problem.

In this semester project, we aim to explore this research field through the production of one scientific and one technical deliverable.

The project description is given in section 2, including the domains addressed in the project, a short explanations on the deliverables produced and the eventual constraints restraining the project initially. Section 3 contains the prerequisites necessary for the completion of the project. The scientific deliverable is then presented in section 4, and the technical deliverable in section 5. And finally, section 6 concludes on the project.

2. Project description

2.1. Domains

Among the domains presented in this section, only the ones related to multimodal machine learning will be discussed in further details in the deliverables of the project. However, more information on each of the domains can be found in the articles cited if interested.

2.1.1. Scientific. The main scientific domain of this project is **multimodal machine learning**, dedicated to machine learning using multiple modalities [1], with the goal to improve the performance of models using additional information from different source modalities. The term "modality" is rather ambiguous in this context, and its definition will often vary, which is why in section 4.3.1, we provide a definition of "modality" as part of the scientific deliverable.

The challenges of multimodal machine learning can roughly be categorized into either representation or fusion. Representation is the problem of encoding data from different modalities and mapping them from one to another, and fusion is the problem of bringing the information from different modalities together and extracting new information from this combination. We discuss some of these challenges in the scientific and the technical deliverable of the project, and the survey article [2] provides further insights on them as well.

This project also touches on the scientific domain of **facial emotion recognition** (FER), which is the task of recognizing emotions in faces. Depending on the context, FER can use data ranging from facial landmarks, over images, to videos [3], depending on the context. Although it is still debated whether computers and programs are adequate tools to evaluate human emotions (e.g. challenges presented in [4], bias issues in [5]), FER and more generally just emotion recognition has interested many AI researchers, as its potential applications are numerous, particularly in human-computer interaction, e.g. [6].

In the technical deliverable, we create neural networks that perform facial emotion recognition.

2.1.2. Technical. The main technical domain of this project is the **Python programming language**. Conceived in the 1980s, it is a popular high level programming language which, alongside R, is used a lot in data science and machine learning.

In this project, we use Python 3.8 to implement the technical deliverable, following object-oriented design patterns. And in particular, we use **Tensorflow**, a python library for machine learning, to implement neural networks. Other libraries used in the project are **numpy** for preprocessing

data, and **matplotlib** and **scikit-learn** and **seaborn** for visualisation and analysis of the results.

2.2. Deliverables

This project produces one scientific and one technical deliverable.

2.2.1. Scientific Deliverables. The scientific deliverable aims to answer the research question:

How did paper [7] normalize multimodal data?

It therefore contains detailed explanations on the model built in the reference article. Additionally, it also provides a definition for the term "modality", which is useful to understand what challenges we are addressing in this project.

2.2.2. Technical Deliverables. The technical deliverable is a multimodal neural network for facial emotion recognition built by combining two unimodal neural networks working on data from different modalities each. The deliverable is implemented using Python and tensorflow, and trained on the FER2013 dataset available online at [8].

2.3. Constraints

We have some constraints for the project which concern the technical deliverable. The first is that we have limited computing power, and using virtual machines is expensive. Therefore, we cannot train big models or use big datasets. Second is that few FER datasets are freely available on the internet, which restricts the choice for the dataset used in this project.

3. Pre-requisites

3.1. Scientific pre-requisites

Minimal preliminary knowledge of multimodal machine learning is required to perform the project. Some background knowledge of neural networks is advised, and incidentally a knowledge of emotion recognition.

3.2. Technical pre-requisites

Basic python programming proficiency is essential for the completion of the project, ideally also some knowledge of the libraries numpy and tensorflow. Additionally, knowledge of object-oriented programming are important.

4. A Scientific Study of Multimodal Data Representation and Learning

4.1. Requirements

The paper "One Model to Learn Them All" [7] explores the perspective of creating a deep learning model capable to solve several different and possibly unrelated tasks. Therefore, they create a multitask model called *MultiModel* and compare its performance on different tasks to the state-of-the-art performance of corresponding task-specific models. While building this model, the main challenge addressed is the multitasking problem. But in order for the model to work on varied tasks, it has to support multiple modalities. For this deliverable, we are focusing on this aspect of their work and define the research question:

How did paper [7] normalize multimodal data?

To answer this question, we set the following functional and non-functional requirements.

4.1.1. Functional Requirements. The scientific deliverable must:

1) present the general architecture of the multimodal neural network built in paper [7].
2) list the modalities supported by this model.
3) explain how data from each modality is standardized
4) explain how standardized data is converted back to modality data
5) Descriptions must explain the logic behind concepts and implementation.

4.1.2. Non-functional Requirements.

1) It must be specified whether the explanations given come from the reference paper, another source or from us.
2) To the best of our ability given our time constraints and lack of knowledge, descriptions in the reference paper must be improved when considered unclear.

4.2. Design

4.2.1. Deliverable structure. To give some context to the explanations on multimodality implementation in MultiModel, the deliverable starts by providing a general presentation of MultiModel, including the tasks it was trained for and the modalities used. This section will also comprise a superficial overview of the model architecture and introduce definitions for terminology specific to the paper.

Once the context is set, the next part presents the structures called "modality nets", which are at the center of our focus, as they are the key to the handling of multiple modalities in MultiModel. Any pre-processing applied on input data is also discussed in this section, as it is directly related with the data processing done by modality nets.

4.2.2. Modality definition. In the previous section, we mentioned the need for some paper-specific terminology definitions in the first section of the scientific deliverable. However, the term "modality", around which the entire

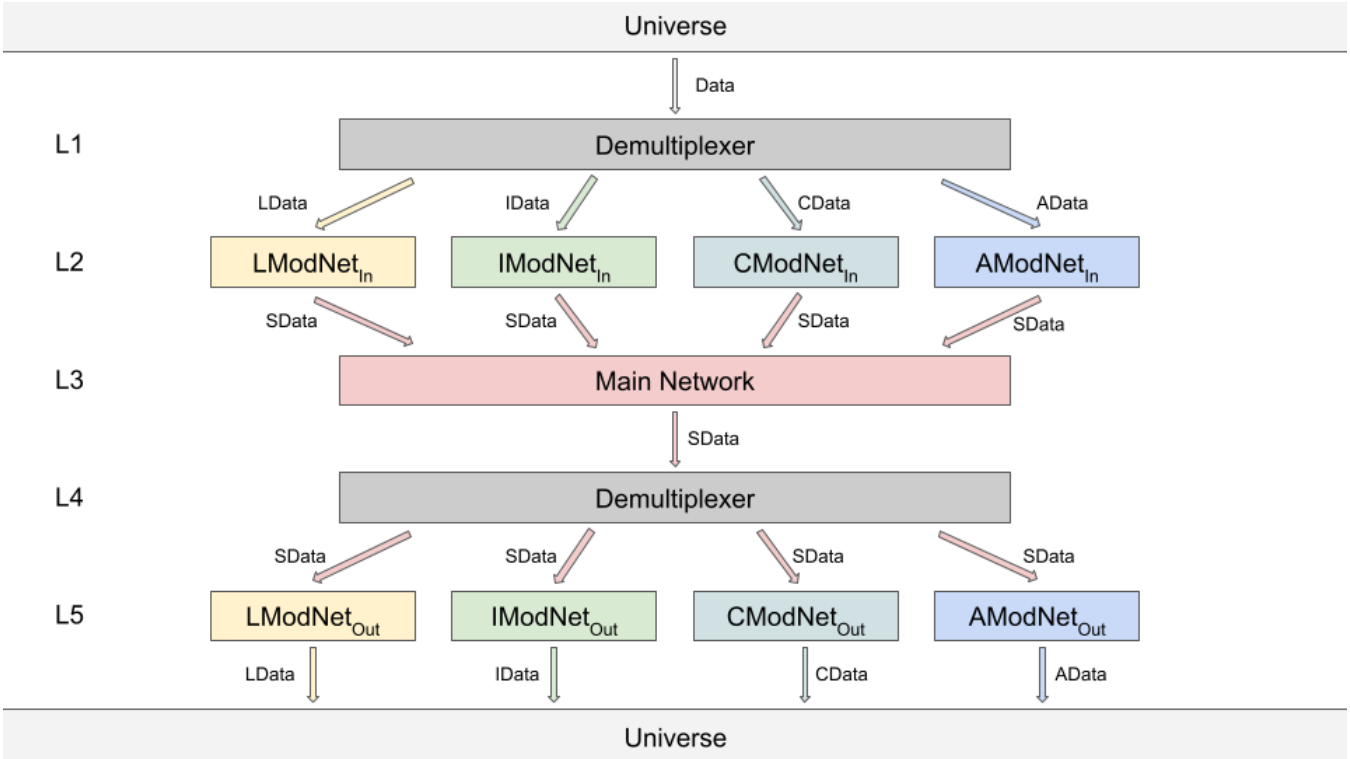


Figure 1: MultiModel architecture.

project revolves, can be subject to multiple interpretations. We therefore need to define it in our context. This definition is introduced at the beginning of the scientific deliverable, and is the one we refer to whenever the term "modality" is mentioned from here onward.

This section also shortly addresses the meaning of "multimodality" in the context of our project, but does not provide a dedicated definition for it.

4.3. Production

4.3.1. Modality and Multimodality. An intuitive definition often given for easy understanding, such as in [1], is that a modality corresponds to "a type of information that we humans perceive with one of our senses". While this definition is simple and works in human-centered contexts, it makes more sense to define modality from a computer-centered point of view in our case, since computers (and computer scientists) are analyzing our data.

From the computer's perspective, information is always encoded in binary, but different representations are used depending on the type of the data. So for instance, images and text will not be stored, parsed and displayed the same way. These are two examples of modalities that we can distinguish in a computer context, but several others exist: sound data [9], EEG signals [10] etc. (the citations are sample studies using data from the mentioned modality) In fact, the definition of modality does not fundamentally change when considering a computer perspective, it still is

"A type of information representation that a computer interprets in a defined way."

The difference is that some information representations may appear to be the same to humans, but different for a computer, and vice-versa.

Take text for instance. For a human, a hand-written text, a picture of the paper with the text, and the same text written on a computer screen will all be part of the same visual modality. But for a computer, the hand-written text can only be accessed via through the picture of the paper and is a bunch of pixels, while the text on the computer screen is a sequence of letters. These two representations are therefore different modalities for a computer, but the same one for a human.

The paper [11] discusses in further details what makes out a modality for humans and computers respectively, and identifies the role of data representation for their definition of "multimodality" for machine learning (ML) tasks. In the latter, assuming that "representation" is analogous to "modality", they consider an ML task to be multimodal when inputs or outputs are composed of distinct modalities[1]. So that means, if we consider an ML task with a single input and a single output modality, it is not multimodal, even if

1. This is a very simplified version of their definition. For their original definition, please refer to the cited paper.

the input and output modalities are different. Only an ML task were input or output modalities respectively differ from each other are considered multimodal.

This description of multimodality can be extended well to models analyzed and trained in our project. So we are considering models that have several input modalities or several output modalities to be multimodal models.

4.3.2. General presentation. The main objective described in paper [7] is to create MultiModel, a deep learning model able to solve tasks from multiple domains. The authors trained MultiModel for 8 different tasks, among which four translation tasks, one object recognition task, one image captioning task, one parsing task and one speech recognition task. Therefore, four different input and output modalities must be handled by the model: Text (or language), image, categorical and audio data.

The model is build mainly to suit the target tasks, but foresees the possibility of adding further tasks in its modular design. Figure 1 depicts the global design of MultiModel. We can see that the authors chose to separate the conversion of data from different modalities (performed by the "modality nets"), and the feature extraction and analysis (done in the main network).

Modality nets are small and computationally lightweight, as their purpose is only to map inputs from different modalities into a standard representation and vice-versa. They are not by any means extracting features from this data, because this is the first role of the main network, the other being analyzing these features to predict an output. This part of the model is complex, involving combinations of mainly attention and convolution blocks.

4.3.3. The "modality nets". As briefly explained in the previous section, MultiModel uses small sub-networks called "modality nets" to handle data from different modalities. More precisely, each modality net converts data of one of the four modalities they use into a standard representation and back. This way, the main network always processes and outputs data within a joint representation space. Conceptually, modality nets can process data in two directions, as pictured in Figure 2:

- The input direction extracts features from the input data and maps them to a standard representation (it does not analyze them - this is the task of the main network)
- The output direction generates an output of the desired modality using the predictions of the main network

However, in their implementation, some modality nets were only implemented to process data in one direction, because the tasks that the MultiModel can perform do not need the other processing direction. The design of modality nets followed two principles:

1) The standard representation does not have a fixed size, because it limits the performance of the model.

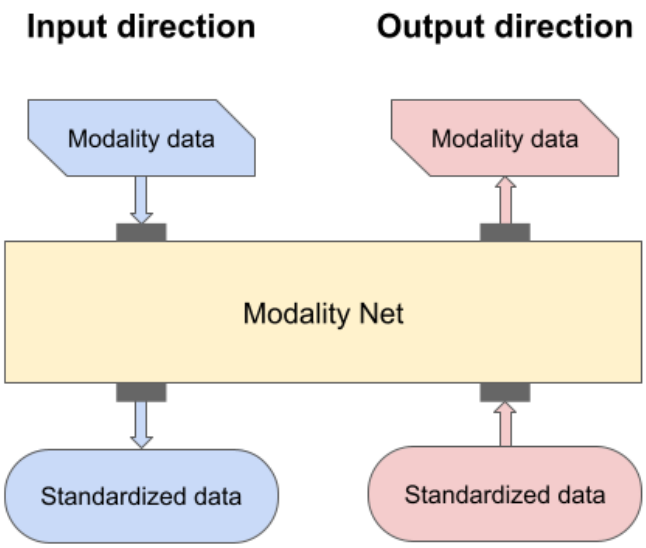


Figure 2: Modality net directions.

2) Modality nets are shared across tasks, such that representations made by the modality nets are generalized and additional tasks can be added easily.

Since there are four modalities (language, images, audio and categories), four modality nets are created.

Language modality net

The language modality net is responsible for extracting features from text data and regenerating text from the output of the model. Therefore, it is used in two directions.

In the input direction, it takes a sequence of tokens as input, so the data must be tokenized in the pre-processing phase. The authors of our reference followed the approach described in the paper [12] to tokenize the text. This approach was developed initially for machine translation tasks as a solution to improve the translation of rare words, and consists of dividing words into subunits (as opposed to keeping the complete words or stemming them for instance). The intuition behind this approach is that a number of potentially rarer words can be understood and translated using the fact that they have similar origins (e.g. "claustrophobia" in English and "Klaustrophobie" in German), or that they share the same morphology (e.g. "solar system" and "Sonnensystem" (i.e. "Sonne" + "System")) in both source and target languages. This tokenization technique is empirically proven in [12] to improve machine translation results, especially on rare words.

We suppose that this approach was chosen because it provides an effective solution to the issue of varying language vocabularies and word scarcities that the language modality net must handle. This theoretically enables to use this modality nets for any text in any language containing any word of the language's vocabulary, which is needed for

MultiModel. In practice, the implementation of the data preprocessing by the authors of [7] uses 8k subword units to tokenize their text data.

The tokenized sequence is then mapped using a learned embedding to the standard representation, which is interpretable by the main network.

In the output direction, the modality net performs a linear mapping of the output of the main network, then applies the softmax activation function to obtain a probability distribution over the token vocabulary that sums up to 1 (therefore predicting a single token).

Image modality net

The image modality net is only implemented in the input direction. Therefore, it is responsible for extracting features from image data and map them onto the standard representation.

For this task, the authors created a model similar the one in the approach "Xception" described in [13]. Xception is itself based on the broader "inception" approach, that partially decouples the extraction of cross-channel correlations and spatial correlations in images by extracting the cross-channel correlations first, then spacial features within each cross-channel correlation space, and concatenating all extracted features. Xception is a stronger version of inception, where the extraction of cross-channel and spatial correlations is entirely decoupled.

No reason are given for the choice of this approach, but we can assume that it enables faster and possibly more accurate extraction of relevant elements in image data than traditional convolution networks.

Audio modality net

Just like the image modality net, the audio modality net was only implemented in the input direction and uses a very similar architecture, so we do not describe it again.

Categorical modality net

The categorical modality net is only implemented in the output direction and is responsible for mapping standardized data to categories.

First, the 1-dimensional standardized data is reshape into the same shape as the input modality (either stays 1D, or becomes 2D), then it is processed with a layer structure similar to the exit flow used in study [13].

The authors of our reference paper [7] do not explain their choice here either, but we can suppose it is chosen to adapt the merging of spatial and temporal dimensions well, because a special layer is used for this purpose in the architecture of this modality net.

4.4. Assessment

To assess the quality of the scientific deliverable, we verify whether the requirements set in section 4.1 are fulfilled.

We start with the functional requirements. The first point is to present the general architecture of MultiModel, which we do in section 4.3.2. In addition to being listed in the section mentioned just before, each modality, is also detailed a little at the beginning of each subsection in section 4.3.3, so the second requirement is fulfilled too. Section 4.3.3 is however primarily explaining how modality data is converted into standardized data and vice-versa by each modality net (and therefore for each modality), fulfilling the next two requirements. Finally, we see that the definition of modality, explanations on multimodality, the final design of MultiModel presentation and the implementation choices are all explained, giving reasons, thought processes and logic explanations. This fulfills the last functional requirement, so therefore, all functional requirements are met.

While the functional requirements were assessed one by one, we are assessing the fulfillment of the non-functional requirements section by section, as it is easier to follow the relationship between content of the section and requirements assessed when doing it this way.

In section 4.3.1, we includes references when ideas are taken from other sources, but when no references are indicated, it may sometimes be unclear whether the information given are our own thoughts or generally accepted concepts. This means that the first non-functional requirement is only partially fulfilled. The second point does not apply to this section, as it only refers to descriptions of the reference paper [7].

Next comes section 4.3.2. Since the paper on MultiModel is cited and information is strictly factual, we consider it obvious that the entire section gives information taken directly from the reference article, so the first point is fulfilled. In this section, we also consider all descriptions of the reference to be clear, which meets the second requirement. Only the terminology was slightly adapted, using the term "main network" instead of the original term "body" for the part of the model corresponding to layer 3 in Figure 1.

Finally, we assess the non-functional requirements for section 4.3.3. Since the reference paper does not justify much of the design choices they make, many of the explanations in this section are deduced ourselves, meaning we improve the original information. And whenever we do so, we specify that it is our supposition only. So, both requirements are fulfilled for this section.

Throughout the sections, we only have one requirement that we consider to not be fully met for one of the sections. Considering that we have 2 requirements and three sections, we can safely say that globally, all non-functional requirements are fulfilled.

5. A Technical Implementation of A Multimodal Model for Facial Emotion Recognition

5.1. Requirements

In the technical deliverable, the aim is to create and train a multimodal model for facial emotion recognition, to explore the challenges of multimodal machine learning. Therefore, we define the following functional and non-functional requirements:

5.1.1. Functional requirements.

1) Construct one unimodal neural network performing emotion recognition on face images (NN1).
2) Build another unimodal neural network, which is performing emotion recognition using facial landmarks (NN2).
3) Create a multimodal neural network performing emotion recognition using both face images and facial landmarks (Final NN).
4) Build two versions of the final NN:
   - Decision-based: Fuses the decisions of NN1 and NN2 to obtain its decision.
   - Feature-based: Uses the features extracted by NN1 and NN2 to reach a decision.
5) Either take all data used to train the neural networks from the FER2013 dataset, or generate it using it.

5.1.2. Non-functional requirements.

1) All NNs must be built and trained using the Python programming language.
2) All NNs trained must be saved persistently, as they belong to the technical deliverable.
3) NN1 and NN2 must have an overall accuracy of at least three times random accuracy, i.e. at least $3/7 = 42.8\%$.
4) The accuracy of the final NN must be better than the accuracy of both individual NNs composing it.
5) The NNs must not be overfitted.
6) The NNs must complete their training within less than 1 hour all together with the GPU *NVIDIA GeForce GTX 1650* we use for their training.

5.2. Design

5.2.1. Data exploration and generation. We use the FER2013 dataset [8] available online for free, built for a facial expression recognition (FER) challenge in 2013 [14]. It contains 35887 grayscale labeled images of site 48x48, categorized into one of seven emotion categories: angry, disgust, fear, happy, sad, surprise and neutral. Figure 3 shows two sample images from the dataset with their labels.

FER2013 contains different amounts of images in each of the category, as is shown on the histogram in figure 4.

It is unknown whether this imbalance is due to a natural imbalanced distribution of emotion expression in humans,

Figure 3: Two sample images of the dataset FER2013.

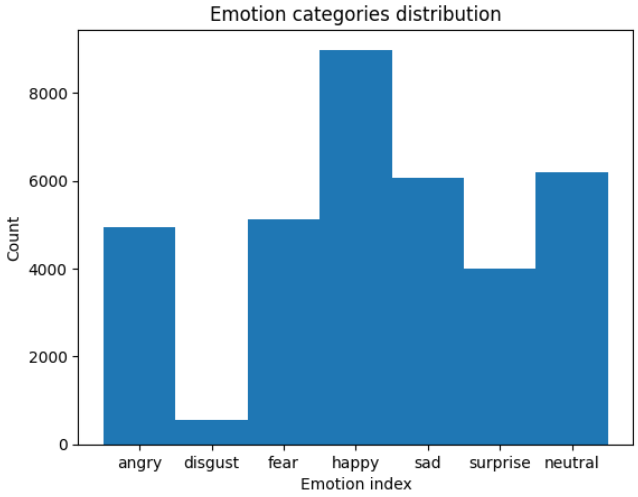


Figure 4: Emotion categories distribution.

but we know for sure that it is problematic for training neural networks. This is why to counterbalance this difference during training, we compute a weight associated with each emotion category. We choose to compute the weights by dividing the highest count (category happy) by the count of the given category. The obtained results are presented in table 1.

Index	Emotion	Count	Weight
0	angry	4953	1.815
1	disgust	547	16.433
2	fear	5121	1.755
3	happy	8989	1.0
4	sad	6077	1.479
5	surprise	4002	2.246
6	neutral	6198	1.450

TABLE 1: Count and weight for each emotion category

These weights are suitable for our purpose, because the penalty they are inflicting to the model when it guesses a category wrong is anti-proportional to the amount of data we have in that given category.

The dataset FER2013 is suitable to train the NN1, which is working with images. NN2 however needs a dataset of

facial landmarks, and instead of using a separate dataset, we generate the data we need from the FER2013 dataset by extracting the facial features of each image and saving them in a separate file. For this task, we use a model from the dlib python library that is available online [15]. Figure 5 shows two sample detection results (the red marks on the images).

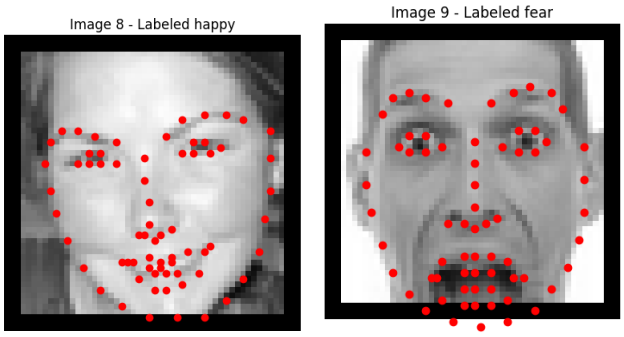


Figure 5: Two sample landmarks detection.

Since each feature point corresponds to a specific facial landmark, we know exactly which ones belong to which part of the face, so we save each part of the face as a separate feature: right eye, left eye, right brow, left brow, nose and so on. We chose to refer to this new dataset as the "FER2013 landmarks dataset".

Note that the FER2013 landmarks dataset is unbalanced in the exact same way than the original FER2013, so the weights computed for the FER2013 categories also hold for our new dataset.

5.2.2. NN1 - ER with face images. The first neural network, which we call NN1, is performing emotion recognition on plain face images from the FER2013 dataset. We use 70% of it as training data, 20% as validation data and 10% as testing data.

NN1 is performing image analysis, so we create a convolutional neural network (CNN) for this task. Figure 6 shows the architecture of the model conceived for it. Of course, we start with an input layer of the dimension of the images. Then, following the references [16] and [17], we stack three groups of each one 2D convolutional layer with relu activation function, followed by one max pooling layer. The figure specifies the hyperparameters for each layer (i.e. number of neurons, kernel size and/or stride, depending on the layer type). This structure is made to extract features from the data. After these groups, we flatten and add two dense layers to group extracted features, and finally add an output layer of seven neurons for the seven categories.

The last two layers do not use relu as activation functions. The output layer uses softmax, to obtain a probability distribution that sums to 1 and therefore have one clearly selected category [18]. The second-to-last layer on the other hand, uses sigmoid, which gives a probability distribution that does not necessarily sum up to exactly 1 [18]. Therefore, this layer is expressing the presence

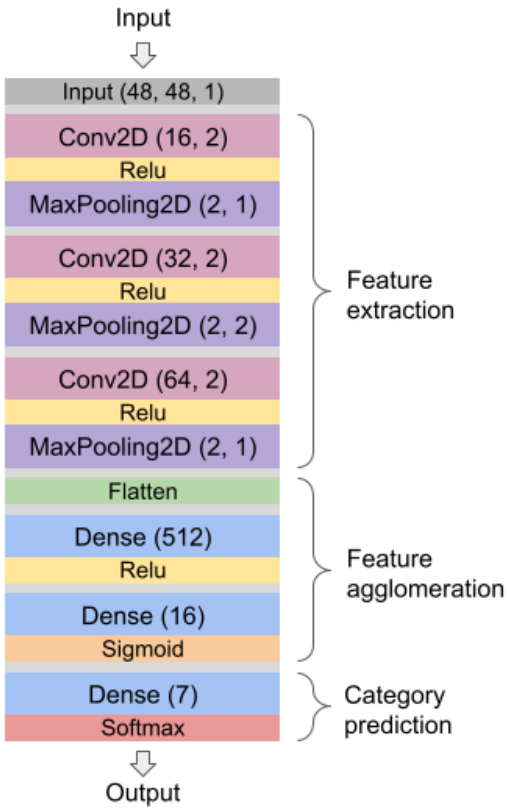


Figure 6: NN1 architecture.

or absence of 16 features or feature combinations that the CNN has extracted. This detail is important for the building of the final neural network, as two versions were created: one using the outputs of each individual NN (i.e. the output layers), and the other the extracted features (i.e. the second-to-last layers of each NN).

For training, NN1 is compiled using the Adam optimizer, because Adam is considered to be the best optimizer in general [19]. The initial learning rate used is 0.0001, and it is decreased by a factor of 0.2 after 10 epochs, so that the NN is tuned with more precision. In total, NN1 is trained over 16 epochs, with a batch-size of 32.

5.2.3. NN2 - ER with facial landmarks. The second neural network, called NN2, is trained on the FER2013 landmarks dataset to recognize facial emotion from specific landmarks position. Like we do for NN1, we use 70% of the dataset as training data, 20% as validation data and 10% as testing data. In fact, we are using the same portions of the dataset respectively than the partitions for NN1, such that both individual networks are trained om data referring to the same images.

NN2 is analyzing vectors corresponding to points which together form the outlines of individual face parts. Our goal for this neural network is to extract features from both individual face parts and their combinations. In theory, a

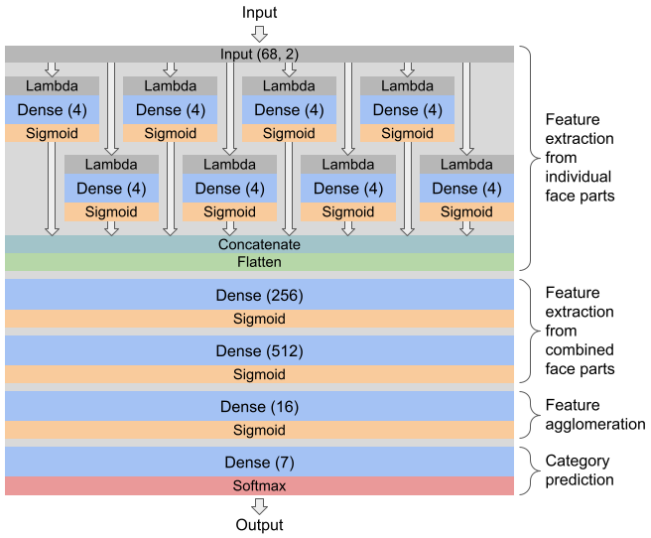


Figure 7: NN2 architecture.

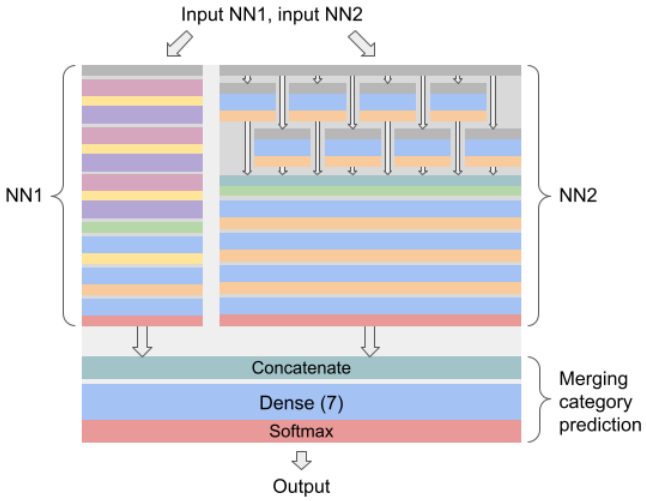


Figure 8: Final NN architecture (decision-based).

NN with several fully connected dense layers is sufficient for this task, but to help the NN with the task of extracting features from the individual face parts, we create a more sophisticated architecture, depicted in figure 7.

The input of 68 (x, y) coordinates of points is first split into 8 parts, corresponding each to one face part (e.g. lips, right eye, jaw), using lambda layers. Each of these parts is then processed in a small dense layer of 4 neurons, activated using sigmoid. The extracted features are then concatenated and flattened, before being fed into two dense layers that are extracting features from the combination of the face parts. These also use the sigmoid activation function to be able to extract as many features as necessary.

Finally, the two last layers are the same as the last two layers of NN1 and serve the same purpose: agglomerate features into 16 feature combinations and predicting the category.

Concerning the training, NN2 is also compiled using the Adam optimizer, this time with an initial learning rate of 0.001. The learning rate is then reduced dynamically during the training by a factor of 0.2 every time the validation loss plateaus over 5 epochs. This way, the model is tuned more precisely for the remaining epochs. In total, NN2 is trained over 16 epochs with a batch size of 32.

5.2.4. Final NN. The final neural network, short final NN, is build through the combination of the pre-trained NN1 and NN2. The neural networks are combined using so-called fusion layers. The fusion layer is obtaining the outputs of layers of NN1 and NN2 and combine their results to output a decision for the final NN. In this project, we build two versions of final NN: a decision-based and a feature-based version.

The architecture of the decision-based final NN is shown in figure 8. As we can see, the outputs of NN1 and NN2 are simply concatenated and then directly mapped onto a dense output layer with softmax that predicts the emotion category.

The decision-based NN is compiled using the Adam optimizer with a constant learning rate of 0.0005 and trained for 16 epochs with a batch size of 32.

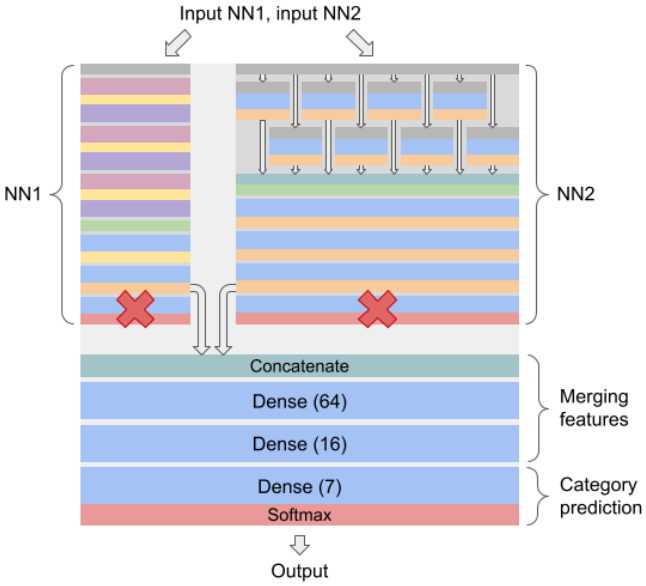


Figure 9: Final NN architecture (feature-based).

The feature-based version on the other hand does not use the predictions of NN1 and NN2 at all. Instead, it concatenates the final features extracted by NN1 and NN2 in their second-to-last layers, then processes these features to extract new ones using dense layers with no activation function. Finally, a dense layer with softmax predicts the emotion category.

This version of the final NN is compiled using the Adam optimizer with a constant learning rate of 0.00005, which

is really low to allow for precise tuning, and trained for 16 epochs with a batch size of 32. Note that the parts of NN1 and NN2 reused in the architecture of the final NN are not fine-tuned during the training of the final NN, only the additional layer(s) are trained in this phase.

5.3. Production

The production section is centered around the implementation of the deliverable, following the design presented in section 5.2. To be more precise, we are covering the implementation all 3 models, but not of the data generation in subsection 5.2.1, as it is not important for the deliverable.

This section starts with subsection 5.3.1, where general information on the code and its structure are given, while the next subsections are presenting the implementation of each class mentioned.

5.3.1. Code structure. We used python to implement and train all neural networks with the tensorflow library and the numpy package. The matplotlib package, alongside the seaborn and scikit-learn libraries, were used to visualize data, plot graphs and create confusion matrices. The dlib library is used to generate part of the dataset.

The code for the technical deliverable is available on GitHub[2] and the trained neural networks (NNs) can be downloaded from google drive[3].

Figure 10 shows the class diagram of the code implementation of the technical deliverable. It only includes the most relevant methods and attributes for the deliverable goals. The operations performed by each individual method of each class are detailed in the upcoming sections. For simplicity, the arguments of the methods will be omitted when describing their features.

5.3.2. NN Model implementation. The NN model is the superclass used for all neural networks built and trained in this project. It provides a default implementation for all methods used to initialize, load, train and save the model, as well as for the generation of graphs and confusion matrices. Some of the implemented methods are meant to be overridden by the subclass implementation, because they inevitably differ from one NN to the other:

- loadData(): loads the dataset and splits it into training, validation and testing sets.
- createArchitecture(): initializes the layers of the neural network.
- prepareModel(): instantiates the epochs and batch size, and more importantly the callbacks if any.

The implementation of these three methods is described for each NN in their corresponding sections in this report. Additionally, some other methods can be overridden if needed and when this happens, these new implementations are also detailed in the corresponding section.

2. https://github.com/IrisBICS/BSP6-Technical-deliverable
3. https://drive.google.com/drive/folders/1tRyeVec0-Ih8gHUIzojiWgUEJAR2AM6l

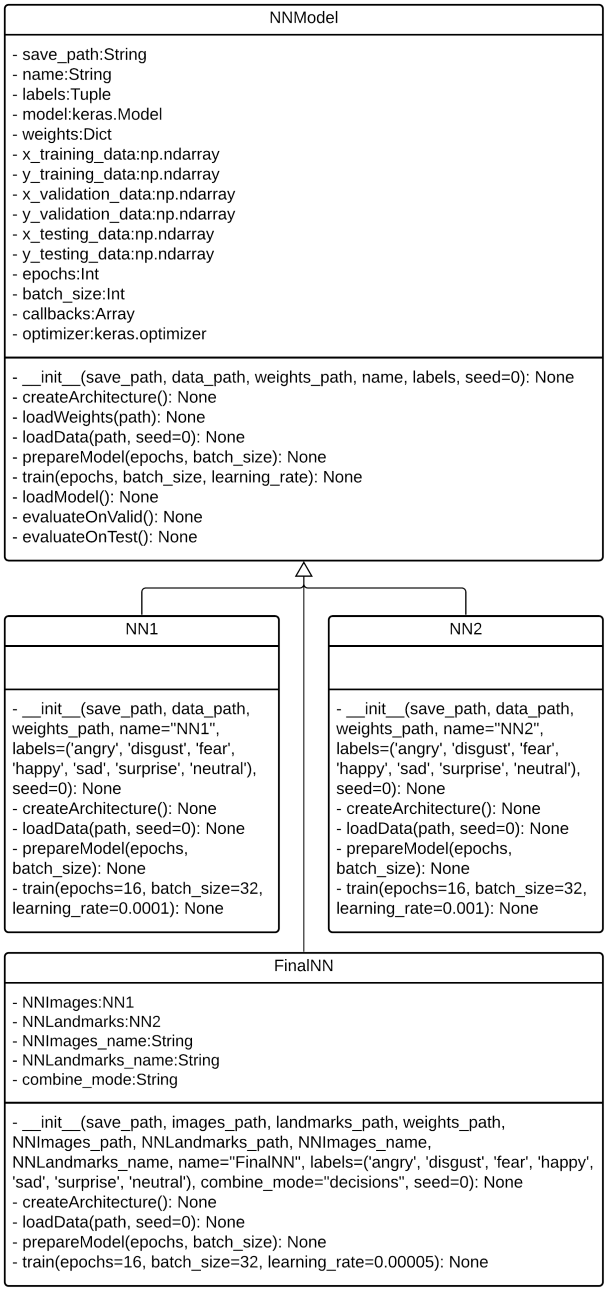


Figure 10: Class diagram of the technical deliverable code.

In this section, we are describing the default implementation the methods listed in figure 10 that do not require to be overridden.

__init__(): Listing 1 shows the implementation of the __init__() method. It creates the object and instantiates its name, saving path and labels (names of categories). It then calls loadWeights() and loadData() to ensure that the model will always have the weights and data available, no matter whether it will be trained or loaded from saved files.

```
def __init__(self, save_path, data_path, weights_path,
     name, labels, seed=0):

    self.save_path = save_path
    self.name = name
    self.labels = labels

    self.loadData(data_path, seed=seed)
    self.loadWeights(weights_path)
```

Listing 1: __init__() method of the NNModel superclass.

loadWeights(): Loads the weights computed for each of the categories.

train(): Listing 2 shows the implementation of the train() method. It first calls createArchitecture() to instantiate the layers of the model. Then, it calls prepareModel() to perform the final preparation steps, before compiling the model and calling the fit() method of tensorflow models to train it. Once training is done, the neural network architecture and weights are saved at the save location of the NN specified during initialisation of the object. The history of the training is also saved there.

```
def train(self, epochs, batch_size, learning_rate):

    self.createArchitecture()
    self.prepareModel(epochs, batch_size)

    self.optimizer = Adam(learning_rate=learning_rate)
    self.model.compile(loss="categorical_crossentropy",
     optimizer=self.optimizer, metrics=["accuracy"])

    self.history = self.model.fit(self.x_training_data,
     self.y_training_data, batch_size=self.batch_size,
     epochs=self.epochs, verbose=1, validation_data=(self.
     x_validation_data, self.y_validation_data), callbacks
     =self.callbacks, class_weight=self.weights)

    self.__saveModel()
```

Listing 2: train() method of the NNModel superclass.

Note that train() allows for some flexibility in the hyperparameters definition, but optimizers, layers and callbacks are defined statically in the implementation of the classes to simplify method calls and limit the number of parameters passed.

loadModel(): Loads the neural network architecture and weights from files in the save location specified during the object initialisation, and compiles it such that it is ready to use. Also loads the training history.

Note that loadModel() should only be called instead of train() and vice-versa; These methods are mutually exclusive.

evaluateOnValid() and evaluateOnTest(): Evaluates the model on the validation and testing data respectively.

5.3.3. NN1 implementation. We use the NN model described in section 5.3.2 as basis for our model object and create a subclass NN1 that overrides three methods: loadData(), createArchitecture() and prepareModel().

loadData() is loading and formatting the images of the FER2013 dataset, shuffles and splits them into training (70%), validation (20%) and testing sets (10%). To be able to reproduce this data splitting in NN2 and have the exact same data partitioning when training the two NNs, we seed the shuffling of the data.

The createArchitecture() method is defined with the code in listing 3. Concretely, it initializes the NN1 model described in section 5.2.2 using tensorflow layers.

```
def createArchitecture(self):

    inp = Input((48, 48, 1))

    hidden = Conv2D(16, 2, activation='relu')(inp)
    hidden = MaxPooling2D(2, 1)(hidden)
    hidden = Conv2D(32, 2, activation='relu')(hidden)
    hidden = MaxPooling2D(2, 2)(hidden)
    hidden = Conv2D(64, 2, activation='relu')(hidden)
    hidden = MaxPooling2D(2, 1)(hidden)
    hidden = Flatten()(hidden)
    hidden = Dense(512, activation='relu')(hidden)
    hidden = Dense(16, activation='sigmoid')(hidden)

    out = Dense(7, activation='softmax')(hidden)

    self.model = Model(inputs=inp, outputs=out)
```

Listing 3: createArchitecture() method for NN1.

Finally, prepareModel() defines a learning rate scheduler callback, which is reducing the learning rate at the 10th epoch by a factor of 0.2, as specified in section 5.2.2. The corresponding code is shown in listing 4.

```
def prepareModel(self, epochs, batch_size):

    def scheduler(epoch, lr):
        if epoch == 10:
            return lr * 0.2
        else:
            return lr

    self.epochs = epochs
    self.batch_size = batch_size

    self.callbacks = []
    self.callbacks.append(LearningRateScheduler(scheduler
     ))
```

Listing 4: prepareModel() method for NN1.

NN1 is trained using the hyperparameters indicated in its design.

5.3.4. NN2 implementation. Like for NN1, we use the NN model described in section 5.3.2 as basis for our model object and create an NN2 subclass that overrides the methods loadData(), createArchitecture() and prepareModel().

loadData() is loading the data from th FER2013 landmarks dataset and formatting it for use in training and evaluation of the NN. The shuffling of the data is seeded by default with the same seed than NN1 and the data is splitted as training, validation and testing into exact same partitions. This way, NN2 is trained with the landmarks corresponding to the images used for training NN1.

The createArchitecture() method, described in listing 5, initializes the NN2 model described in section 5.2.3 using tensorflow layers.

```
def createArchitecture(self):

    inp = Input(shape=(68, 2))

    jaw_inp = Lambda(lambda x: x[:, :17])(inp)
    left_brow_inp = Lambda(lambda x: x[:, 17:22])(inp)
    right_brow_inp = Lambda(lambda x: x[:, 22:27])(inp)
    nose_inp = Lambda(lambda x: x[:, 27:36])(inp)
    right_eye_inp = Lambda(lambda x: x[:, 36:42])(inp)
    left_eye_inp = Lambda(lambda x: x[:, 42:48])(inp)
    mouth_inp = Lambda(lambda x: x[:, 48:61])(inp)
    lips_inp = Lambda(lambda x: x[:, 61:])(inp)

    jaw_dense = Dense(4, activation="sigmoid")(jaw_inp)
    left_brow_dense = Dense(4, activation="sigmoid")(
     left_brow_inp)
    right_brow_dense = Dense(4, activation="sigmoid")(
     right_brow_inp)
    nose_dense = Dense(4, activation="sigmoid")(nose_inp)
    right_eye_dense = Dense(4, activation="sigmoid")(
     right_eye_inp)
    left_eye_dense = Dense(4, activation="sigmoid")(
     left_eye_inp)
    mouth_dense = Dense(4, activation="sigmoid")(
     mouth_inp)
    lips_dense = Dense(4, activation="sigmoid")(lips_inp)

    concat = concatenate(axis=1, inputs=[jaw_dense,
     left_brow_dense, right_brow_dense, nose_dense,
     right_eye_dense, left_eye_dense, mouth_dense,
     lips_dense])
    flatten = Flatten()(concat)
    hidden = BatchNormalization()(flatten)

    hidden = Dense(256, activation="sigmoid")(hidden)
    hidden = Dense(512, activation="sigmoid")(hidden)
    hidden = Dense(16, activation="sigmoid")(hidden)
    dropout = Dropout(0.2)(hidden)
    out = Dense(7, activation="softmax")(dropout)

    self.model = Model(inputs=inp, outputs=out)
```

Listing 5: createArchitecture() method for NN2.

Lastly, the main function of prepareModel(), shown in listing 6, is to define a ReduceLROnPlateau callback which is reducing the learning rate by a factor of 0.2 whenever the validation loss ceases to decrease for 5 consecutive epochs, as required in the design.

```
def prepareModel(self, epochs, batch_size):

    self.epochs = epochs
    self.batch_size = batch_size

    self.callbacks = []
    self.callbacks.append(ReduceLROnPlateau(monitor='
     val_loss', factor=0.2, patience=5, verbose=1, mode='
     auto'))
```

Listing 6: prepareModel() method for NN2.

Here too, NN2 is trained using the hyperparameters indicated in its design.

5.3.5. Final NN implementation. The final NN implementation is a little trickier than the implementation of NN1 and 2, first because it reuses both NN1 and NN2, and second because there are two versions of the model implemented in the same FinalNN class. Additionally to overriding the methods loadData(), createArchitecture() and prepareModel() as required, the implementation of the final NN also changes the __init__() method of the NNModel superclass.

In the __init__() method, presented in listing 7, FinalNN is first instantiating, then loading pre-trained versions of NN1 as "NNImages" and NN2 as "NNLandmarks" (lines 14 to 18). It then sets the trainable parameters of the two NNs to False, such that they are not trained/fine-tuned along during the training of the final NN. Finally, it calls the superclass init method to finalize the instantiation of the FinalNN object.

```
def __init__(self, save_path, images_path, landmarks_path
     , weights_path, NNImages_path, NNLandmarks_path,
     NNImages_name,
            NNLandmarks_name, name="FinalNN", labels=('
     angry', 'disgust', 'fear', 'happy', 'sad', 'surprise'
     , 'neutral'),
            combine_mode="decisions", seed=0):

    self.images_path = images_path
    self.landmarks_path = landmarks_path
    self.weights_path = weights_path
    self.NNImages_path = NNImages_path
    self.NNLandmarks_path = NNLandmarks_path
    self.NNImages_name = NNImages_name
    self.NNLandmarks_name = NNLandmarks_name
    self.combine_mode = combine_mode

    self.NNImages = NN1(self.NNImages_path, self.
     images_path, self.weights_path, name=self.
     NNImages_name, seed=seed)
    self.NNLandmarks = NN2(self.NNLandmarks_path, self.
     landmarks_path, self.weights_path, name=self.
     NNLandmarks_name, seed=seed)

    self.NNImages.loadModel()
    self.NNLandmarks.loadModel()

    self.NNImages.model.trainable = False
    self.NNLandmarks.model.trainable = False

    super().__init__(save_path, None, self.weights_path,
     name, labels, seed, verbose)
```

Listing 7: __init__() method for the Final NN.

The loadData() method reuses the data loaded in the NNImages (NN1) and NNLandmarks (NN2) objects as training, validation and testing data for the final NN.

The createArchitecture() method initializes the final NN model as described in section 5.2.4. Its code is shown in listing 8. The parameter "combine_mode", passed to the object at initialisation time, determines which version of the final NN architecture is used. By default, the decisions of NN1 and NN2 are used (lines 12 and 13), because this version is the least complex, but if the string "features" is passed as value for "combine_mode", the model will use the features extracted by NN1 and NN2.

```
def createArchitecture(self):

    for layer in self.NNImages.model.layers:
        layer._name = "image_" + layer._name
    for layer in self.NNLandmarks.model.layers:
        layer._name = "landmarks_" + layer._name

    if self.combine_mode == "features":
```

```
        combined_out = concatenate([self.NNImages.model.
    layers[-2].output, self.NNLandmarks.model.layers[-2].
    output])
        combined_out = Dense(64)(combined_out)
        combined_out = Dense(16)(combined_out)
    else:
        combined_out = concatenate([self.NNImages.model.
    output, self.NNLandmarks.model.output])

    out = Dense(7, activation='softmax')(combined_out)

    self.model = Model(inputs=[self.NNImages.model.input,
      self.NNLandmarks.model.input], outputs=out)
```

Listing 8: createArchitecture() method for the Final NN.

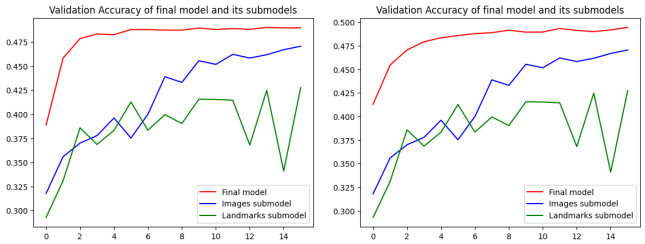
Lastly, the prepareModel() method this time does not define any callbacks to be used during training, as none is required according to the design.

Finally, the final NN is trained using the hyperparameters indicated in its design.

5.4. Assessment

In this section, we start by presenting the results obtained for the final NNs and compare it with the two individual NNs. Then, we discuss the possible improvements that can be made to improve the results. And finally, we see whether the deliverable fulfills its requirements.

5.4.1. Final NN against NN1 & 2. Since we have two versions of the final NN, let us first compare the accuracy obtained with each of them to those of the individual NNs.



(a) Decision-based. (b) Feature-based.

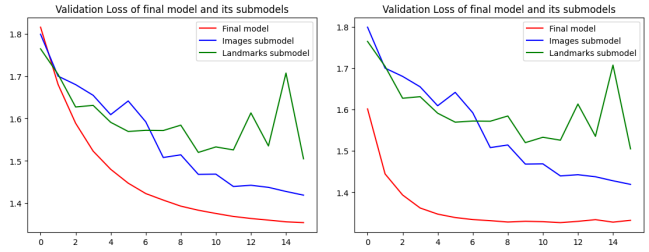
Figure 11: Final NN accuracy comparison with NN1 and NN2.

Figure 11 shows the evolution of the validation accuracy of the neural networks over the training epochs. The red lines stand for the final NNs, the blue ones for NN1 and the green ones for NN2. As we can observe, NN2 performs the worst and has an unstable accuracy over the epochs.

Model	Validation data	Testing data
NN1	47.0%	45.2%
NN2	42.7%	43.9%
Final NN (Decisions)	48.9%	48.0%
Final NN (Features)	**49.4%**	**48.5%**

TABLE 2: Final accuracy of the NNs on the validation and testing data.

In table 2, we see the accuracy of each NN on the validation and testing data respectively. NN1 reaches an accuracy of around 46%, NN2 around 43.5%, the decision-based final NN around 48.5% and the feature-based final NN goes up close to 49%. So, all NNs have an overall accuracy higher than 3 times a fully random accuracy (1/7). However, these accuracies all remains around 20% (or worse) lower than to the best results for the FER2013 challenge, that has the exact same task as our NNs with the same dataset [14].



(a) Decision-based. (b) Feature-based.

Figure 12: Final NN loss comparison with NN1 and NN2.

In figure 12, where we see the evolution of the losses of Final NN compared to those of NN1 and NN2, we observe results which are consistent with the accuracy results we obtained.

Since the feature-based final NN performs a little better that the decision-based one, we will only discuss this version for the final NN from now onwards.

The accuracy and loss give us an indication of the overall performance of the NNs, but confusion matrices enable to look into which categories are detected the best, and percentages of false positives and negatives.

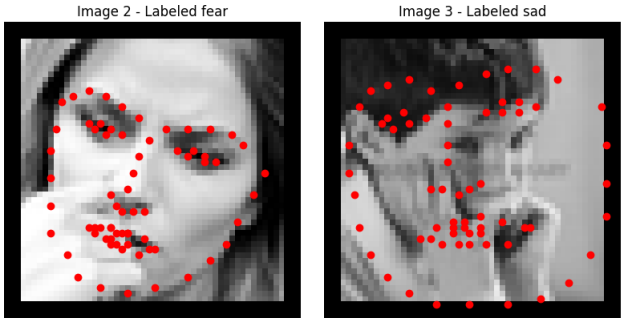
Figure 13 shows the confusion matrices built using the testing data for each of the neural networks. The numbers in the diagonal show the percentages of correct guesses per category. The percentages in the row of a category (except the one on the diagonal) are false negatives of that category, while the percentages in the column of a category are the false positives of that category. For instance, considering the happy row, the percentage in the disgust column is the percentage of happy faces predicted as disgusted ("false disgusted"), while in the happy column, the percentage in the disgust row is the percentage of disgusted faces predicted as happy ("false happy").

A few interesting things can be noted in the confusion matrices of figure 13. First, there are some categories that are identified much better than others in general (disgust, happy and surprise vs. angry, fear and sad). Second, when we compare the confusion matrix for the final NN with the two other ones, we see that the final NN appears to use more features from certain NNs for the prediction of fear (NN1) and disgust (NN2), because the accuracies of the respective individual NN is far better than the accuracy from the other individual NN for this category and close to the one of the final NN. On the other hand, the predictions of the categories angry, sad and surprise are improved

compared to both individual NN's predictions, suggesting that the final NN combines information extracted by both NNs to make an improved prediction in these categories. Finally, we also see that the accuracies of happy appears to be some average of the other's accuracies, and the neutral category is even detected worse than both NNs.

5.4.2. Improvements. Although the results obtained for this deliverable are of sufficient quality to satisfy the requirements, as we will see in section 5.4.3, they remain far from state of the art results such as those from [14], [20]. Due to lack of time and knowledge, we cannot spend more time on improving our datasets, models and training procedures, so instead, we are summarizing some important points identified that can be improved in this section.

FER2013 landmarks dataset



(a) Good landmarks detection. (b) Bad landmarks detection.

Figure 14: Two samples from the FER2013 landmarks dataset.

The FER2013 landmarks dataset, on which NN2 is trained, is noticeably flawed, which certainly negatively impacts the performance of NN2 and therefore also the final NN. Figure 14 shows two samples of the dataset (the red dots on the images). Many pictures give good detection results, like in subfigure 14b, but there are still a significant amount of qualitatively poor detection results, such as subfigure 14a.

In fact, when considering the confusion matrix of NN2 in figure 13b, we can guess that the fear category in particular contains a high amount of bad samples, because NN2 is almost never able to classify fear correctly, possibly because its samples are consistently too close to those of the other categories.

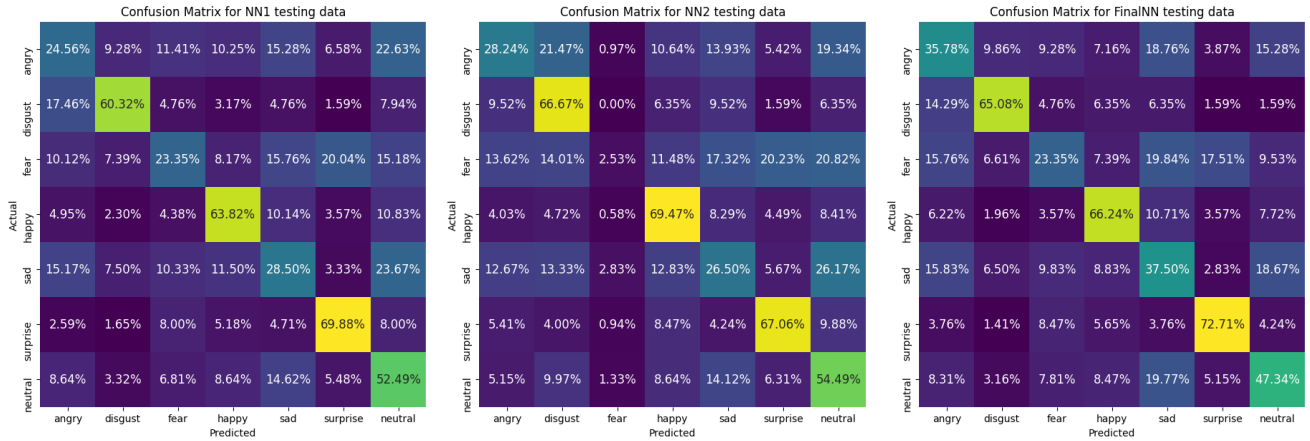
Sadly, we have no way of determining through code which samples are bad, so the only option left is to manually cleaning the dataset from the bad data. It is time consuming, but it would be a good way to improve the results given by NN2.

Model architectures

The choices made for the architectures of NN1, NN2 and the two versions of the Final NN are constraint by both our knowledge of neural networks and time. Numerous intermediate versions of the NNs were built before we came to the final results presented in this report, so the layer types, their structure and the hyperparameters are already improved to the best of our ability based on testing. Still, we are aware that they are far from optimal, and that with more testing and knowledge of NNs, we can reach better results.

It is important to note that when building and training the NNs, another constraint arises: NN1 and NN2 need to have similar accuracies. Otherwise, the model that performs worse is completely ignored in the training of the final NN, which we need to avoid according to our requirements.

So for instance, in one of the intermediate versions of NN1, we reached an accuracy of around 60%, by using



(a) NN1. (b) NN2. (c) Feature-based final NN.

Figure 13: Confusion matrices on the testing data for each of the neural networks.

ResNet50, a pre-trained CNN publicly available. However, we could not reach a similar accuracy with NN2, causing issues in the training of the final NN, because it was only basing its decisions on the features and decisions of NN1, completely ignoring NN2. For this reason, we decided to not use ResNet50 in NN1, resulting in an NN1 downgraded to fit with NN2.

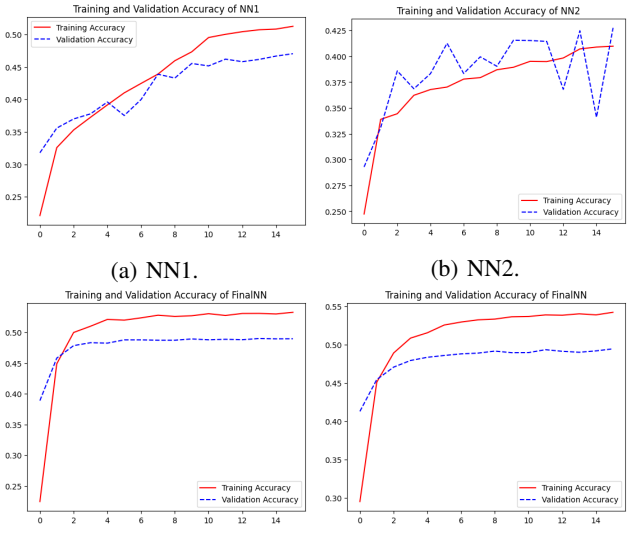
In conclusion, this means that we need to keep the accuracies of both NN1 and NN2 similar when we improve them.

5.4.3. Fulfillment of requirements. The technical deliverable can only be considered a success if its functional and non-functional requirements listed in section 5.1 are fulfilled.

When we look at the technical deliverable, we can easily see that the functional requirements are all met. Indeed, we create NN1, NN2 and both versions of the Final NN using the training data and architectural elements specified in the requirements.

For the non-functional requirements, we know that the points 1, 2, 3 and 4 are fulfilled from the information given in sections 5.3 and 5.4.1.

To assess point 5, we need to compare training and validation accuracies of the NNs over the training epochs, as pictured in Figure 15.



(a) NN1. (b) NN2.

(c) Decision-based Final NN. (d) Feature-based Final NN.

Figure 15: Comparison of training and validation accuracy over the training epochs.

We see in graph 15a that NN1 seems slightly overfitted, as there is a clear gap between training and validation data at epochs 11 to 16, and that the NN consistently performs better on the training data. The gap is small and does not widen over epochs, signifying that the NN is not overfitting increasingly more over training. But we still have some level of overfitting present, which is problematic for the Final NN as we see in a few sentences. NN2 in figure 15b on the other hand does not show particular overfitting signs. The validation accuracy is quite unstable, but the final result in epoch 16 shows that the validation accuracy is a little better than the training accuracy. We could say in this case that a little underfitting is present, but nothing particularly important. Both versions of the final NN in figures 15c and 15d show a little overfitting from epoch 3 all the way up to epoch 16 onwards. The gap between training and validation accuracy also slightly increases over the epochs, indicating that the overfitting is getting worse. We can likely trace this overfitting back to NN1 performances, because the overfitted property of NN1 gets transfered onto the Final NN as soon as the Final NN uses the decisions or features made by NN1 to make predictions, since the training data used for the final NN is the same as for NN1 training.

So, point 5 is only met for NN2, not for NN1 and the two versions of the Final NN.

As for point 6, the time needed to train each NN was not precisely measured, but we are certain that it took less that 45 minutes all together to train all NNs. So, the last non-functional requirement is met.

To summarize, all requirements for the technical deliverable are fulfilled except for non-functional requirement number 5.

Acknowledgment

I, Iris Kremer, would like to thank my project academic tutor Prof. Nicolas Guelfi for his consistent guidance and precious help throughout the project.

I would also like to thank my family and friends for their continuous support.

And finally, as this is my last bachelor semester project, I would again like to thank the BiCS management team for the amazing opportunity that they give us BiCS students to explore and gain knowledge in domains of our choices with these projects.

6. Conclusion

To conclude, this paper presents the sixth bachelor semester project made by Iris Kremer, under the direction of her project academic tutor Prof. Nicolas Guelfi.

In this project, we produce one scientific and one technical deliverable on the general topic of multimodal machine learning. Both are fulfilling nearly all their requirements, so we consider this project to be a success.

Future work for the scientific deliverable can be to analyse the performance results obtained in the reference paper and understand the benefits of their architecture in further details. As for the technical deliverable, a direction to take in the future can be to implement the improvements suggested in section 5.4.2. A second aspect to improve

would be to get rid of overfitting, which would fulfill the non-functional requirement number 5 and of course improve the quality of the deliverable.

References

[1] Tadas Baltrušaitis, Chaitanya Ahuja, and Louis-Philippe Morency. Multimodal machine learning: A survey and taxonomy. *IEEE transactions on pattern analysis and machine intelligence*, 41(2):423–443, 2018.

[2] Chao Zhang, Zichao Yang, Xiaodong He, and Li Deng. Multimodal intelligence: Representation learning, information fusion, and applications. *IEEE Journal of Selected Topics in Signal Processing*, 14(3):478–493, 2020.

[3] Byoung Chul Ko. A brief review of facial emotion recognition based on visual information. *sensors*, 18(2):401, 2018.

[4] Soujanya Poria, Navonil Majumder, Rada Mihalcea, and Eduard Hovy. Emotion recognition in conversation: Research challenges, datasets, and recent advances. *IEEE Access*, 7:100943–100953, 2019.

[5] Lauren Rhue. Racial influence on automated perceptions of emotions. *Available at SSRN 3281765*, 2018.

[6] Agata Kołakowska, Agnieszka Landowska, Mariusz Szwoch, Wioleta Szwoch, and Michał R Wróbel. Emotion recognition and its application in software engineering. In *2013 6th International Conference on Human System Interactions (HSI)*, pages 532–539. IEEE, 2013.

[7] Lukasz Kaiser, Aidan N Gomez, Noam Shazeer, Ashish Vaswani, Niki Parmar, Llion Jones, and Jakob Uszkoreit. One model to learn them all. *arXiv preprint arXiv:1706.05137*, 2017.

[8] Pierre-Luc Carrier, Aaron Courville. Facial emotion recognition dataset 2013. https://www.kaggle.com/c/challenges-in-representation-learning-facial-expression-recognition-challenge/data. Online; accessed 8th of May 2021.

[9] Munaf Rashid, SAR Abu-Bakar, and Musa Mokji. Human emotion recognition from videos using spatio-temporal and audio features. *The Visual Computer*, 29(12):1269–1275, 2013.

[10] Wei Liu, Wei-Long Zheng, and Bao-Liang Lu. Emotion recognition using multimodal deep learning. In *International conference on neural information processing*, pages 521–529. Springer, 2016.

[11] Letitia Parcalabescu, Nils Trost, and Anette Frank. What is multimodality? *CoRR*, abs/2103.06304, 2021.

[12] Rico Sennrich, Barry Haddow, and Alexandra Birch. Neural machine translation of rare words with subword units. *arXiv preprint arXiv:1508.07909*, 2015.

[13] François Chollet. Xception: Deep learning with depthwise separable convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1251–1258, 2017.

[14] Ian J Goodfellow, Dumitru Erhan, Pierre Luc Carrier, Aaron Courville, Mehdi Mirza, Ben Hamner, Will Cukierski, Yichuan Tang, David Thaler, Dong-Hyun Lee, et al. Challenges in representation learning: A report on three machine learning contests. In *International conference on neural information processing*, pages 117–124. Springer, 2013.

[15] Dlib python library. Facial landmarks recognition model from dlib. https://github.com/davisking/dlib-models/blob/master/shape_predictor_68_face_landmarks.dat.bz2. Online; accessed 8th of May 2021.

[16] Sidath Asiri. Building a convolutional neural network for image classification with tensorflow. https://medium.com/@sidathasiri/building-a-convolutional-neural-network-for-image-classification-with-tensorflow-f1f2f56bd83b. Online; accessed 9th of May 2021.

[17] Udeme Udofia. Basic overview of convolutional neural network (cnn). https://medium.com/dataseries/basic-overview-of-convolutional-neural-network-cnn-4fcc7dbb4f17. Online; accessed 9th of May 2021.

[18] Rachel Lea Ballantyne Draelos. Multi-label vs. multi-class classification: Sigmoid vs. softmax. https://glassboxmedicine.com/2019/05/26/classification-sigmoid-vs-softmax/. Online; accessed 9th of May 2021.

[19] Sebastian Ruder. An overview of gradient descent optimization algorithms. https://ruder.io/optimizing-gradient-descent/index.html#whichoptimizertouse. Online; accessed 9th of May 2021.

[20] Juan Luis Rosa Ramos. *Deep learning for universal emotion recognition in still images*. PhD thesis, UPC, Facultat d'Informàtica de Barcelona, Apr 2018.

7. Appendix