Computer Vision - Assignment 2 - CNNs

Sacha L. Sindorf (i6259085)
Dimitrios Gagatsis (i6258651)

May 2021

1 Introduction

This assignment aims to give a solution to the emotion recognition task, when is it defined as a classification task. In particular, we can analyze and detect the emotions from a given data set by using deep learning models and creating a convolution neural network. However, creating a convolutional neural network requires experimentation to find the best-evaluated architecture that will give the best accuracy for the given task.

2 Data preparation and inspection

The data has been taken from (*FER2013 dataset*, n.d.). Some details on this data was given at (*FER2013 dataset discussion*, n.d.). It contains 48x48 grayscale pictures labeled as (0=Angry, 1=Disgust, 2=Fear, 3=Happy, 4=Sad, 5=Surprise, 6=Neutral).

There are 3 sets in this data: *Training*, *PublicTest*, and *PrivateTest*. PublicTest was used in a Kaggle competition to evaluate submissions. For the project's purpose the sets Training and PrivateTest will be used.

The dataset is in csv format. In order to get faster load times and compact storage, this format is translated to numpy-arrays and saved as numpy-binaries. The sets are split in X-images, and Y-labels: *X_training.npy*, *X_privatetest.npy*, *X_publictest.npy*, *Y_training.npy*, *Y_privatetest.npy*, *Y_publictest.npy*.

The labels are translated to one-hot encoded arrays of 7 bits length. The output layer of the neural network will consist of 7 nodes with a *SoftMax* on them. Consequently, every NN output node will give a value corresponding to a likelihood that an image input belongs to a class.

An inspection of the data can be seen in Figure 1 and Figure 2, where the histograms are plotted. The first observation is that not all classes are being represented equally. The second observation is that training and test data have similar distributions.

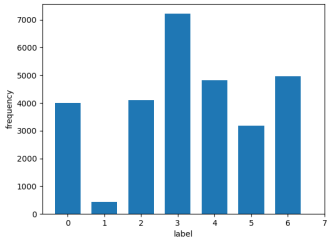

Figure 1: Histogram training labels

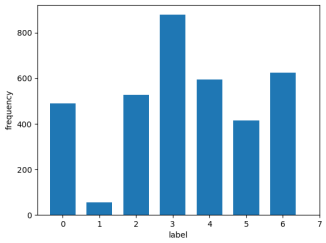

Figure 2: Histogram privatetest labels

3 Training set up

It was decided not to use the torchvision package as suggested in the assignment. The dataloader expects a certain input format and it would take longer to study this than to code a loop running through the data as batches. Batching is necessary as the data is too large to be trained upon as one array.

The training structure was build by using a small portion of data and a small architecture (ConvNet0 in Appendix A). Followed by debugging until a typical loss development as in Figure 3 appeared. The reason for making making the problem smaller first, is make faster debugging cycles.

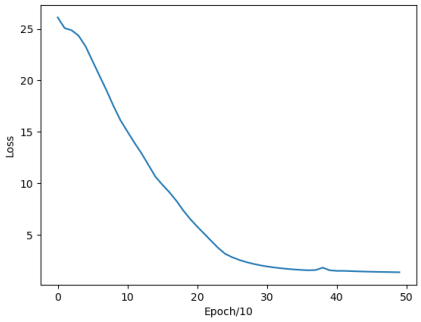

Figure 3: Loss in training over epochs.

With the full datasets and a large architecture (ConvNet1 in Appendix A) some experiments were run to get a feeling for quality and acceptable run times for a laptop-at-home development environment. It was decided to settle for the settings in Table 1 and tune model quality further based on choosing different CNN architectures.

normalization	epochs	loss function	optimizer	batch size	learning rate
/255.0	200	MSELoss	Adam	128	0.0001

Table 1: Training settings.

4 Model development

The general structure of the architectures uses in this project, is 3 convolution layers and 2 fully connected layers. A convolution layer consists of modules: *Conv2d*, *BatchNorm2d*, *ReLU*, *Dropout2d*, *MaxPool2d*. A fully connected layer consists of modules: *Linear*, *BatchNorm1d*, *Dropout*, *ReLU*. It was hinted to stay below 5 layers. The intuition for 3 convolution layer and 2 fully connected layers

is, to use the convolution layers as analyzers on the images and to use the fully connected layers for classification. Of course, how these layers are used exactly is not decided by the developers, but determined by the training process.

The model development can be followed in Appendix A. From top to bottom it is a history of different trials. Although the goal was to reach the best accuracy, this metric was not the main guidance for making design choices. The loss metric was used for this, calculated on a trained model, both for test and training data. For training data a random selection was made of equal length as the test data. See Table 2 for a development of the quality metrics.

Model	0	1	2	3	4	5	6
Test Accuracy	45.8	61.4	59.1	59.2	57.3	61.2	61.3
Test Loss	2594.5	1883.8	2013.6	1998.6	2082.8	1942.7	1962.5
Training Loss	1925.6	15.2	16.6	11.6	28.2	1119.3	1133.7

Table 2: Score metrics during development.

Model-0 is used to set up the development flow and only plays a small role in the development history. From Model-1 on, Test Loss and Training Loss were compared and this reasoning was used: model is overfitted to training data if training loss is very small compared to test loss.

The strategy was then to reduce the dimensions of the CNN to force it to generalize further, but at Model-4 the Accuracy started to suffer, while still not doing a good job at generalization. Then Model-3 was used as basis for Model-5, but the dropout was increased from 0.1 to 0.4. This resulted in a model that generalizes and still has a good accuracy. Model-1 does have a better accuracy, but it is has many more weights, making it more expensive and slower to train.

The confusion matrix for Model-5 can be seen in Table 3. It shows to which classes the testdata gets labeled.

	0	1	2	3	4	5	6
0	238	4	33	46	90	12	68
1	14	20	3	8	6	2	2
2	53	1	193	41	98	56	86
3	20	0	16	756	36	13	38
4	53	1	58	57	299	7	119
5	10	0	32	34	17	300	23
6	36	0	18	66	106	6	394

Table 3: Confusion matrix model 5.

The largest class getting confused by Model-5 is class 4 being predicted as class 6. This happened 119 times. Some examples are depicted in Figure 4. These are 'Sad' faces predicted as 'Neutral'. Their accompanying output node scores do show that label-6 is highest, but not dominating. Some pictures also have an increased likelihood for label-4: 'the model doubts'.

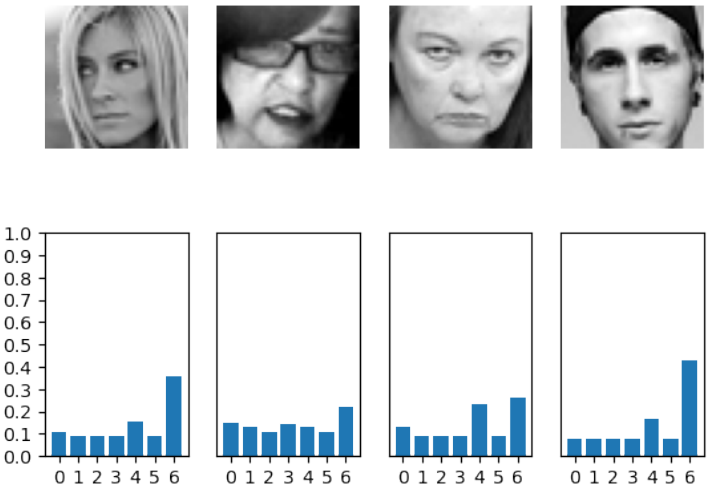

Figure 4: Model on test data: ’4 predicted as 6’

One good go on-and-on optimizing the architecture and training settings, but it was decided to stop here and try something else: data augmentation.

5 Data augmentation

More training data can be made by artificially changing training data and add to it. Change should be in such a way that the label still covers the image. The ideas is to produce more accurate models by using more, cheaply acquired, training data.

In this experiment the aim is reduce the ’4 predicted as 6’ group. For this the training data is fed to the trained model, and images from this same confused group are extracted from the training set. See Figure 5 for examples from the training data. By offering more training data from this category, the model might learn to distinguish better between classes 4 and 6.

The new, augmented, training data are vertically mirrored versions of the old data, see Figure 6. This data is inserted to the training data set at randomly selected positions. Unfortunately, this group is only 149 units large. Apparently training managed to suppress the amount if mistakes here.

Other methods of augmentation were considered. As the images are closely cut around the faces there is not much room for shifting or rotation. First explore how vertically mirrored training data impacts the results.

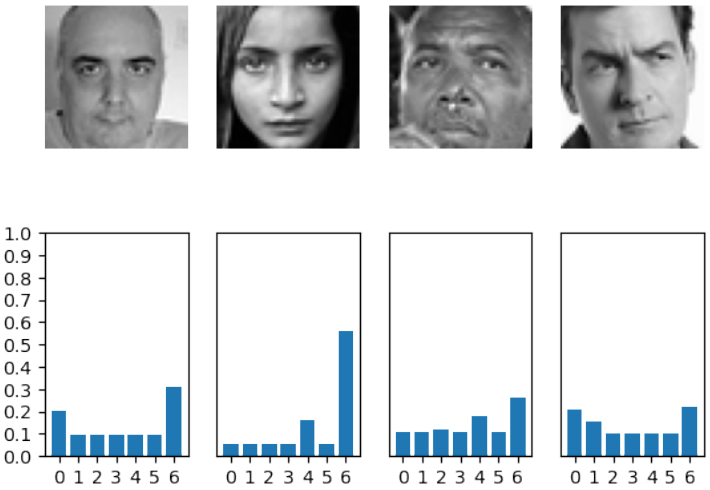

Figure 5: Model on training data: '4 predicted as 6'

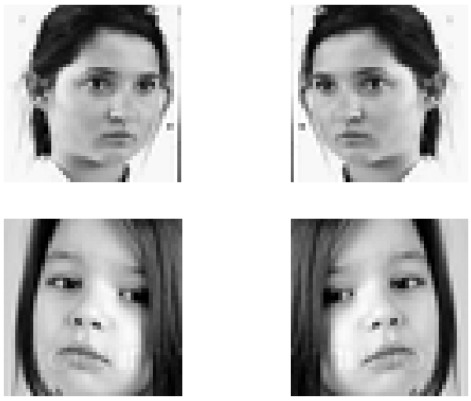

Figure 6: Augmentation training data from category '4 predicted as 6'

The result can be found in Table 2 at Model-6. The test accuracy increased a bit. The confusion matrix for Model-6 is shown in Table 4. Reducing the mislabeled group from 119 to 116 is not a significant success.

The extra training data was not that much to have serious impact. Perhaps other categories could have been extracted from training data as well, for instance everything labeled as 4 or 6. The output node values could have been studied - where likelihood for 4 is close to 6: add to augmentation set.

	0	1	2	3	4	5	6
0	249	2	37	46	79	12	66
1	14	22	7	5	3	2	2
2	62	2	196	41	94	61	72
3	21	0	16	764	34	12	32
4	72	0	48	69	276	13	116
5	17	0	33	28	10	307	21
6	48	0	22	64	96	9	387

Table 4: Confusion matrix model 6.

More quality analysis on Model-6 can be found in Table 5. Lowest recall is for label 2. This

means the model finds it most difficult to correctly identify facial expression 'Fear'. Lowest precision is for label 4. This means the model tends to wrongly identify faces as having the 'Sad' expression. The f1-score is the harmonic mean of precision and recall. For improving the scores further one could think of measures to improve identification of label 2.

Important to notice are the high scores on label 3. As can be seen in Figure 1, this is also the label overly represented in the training set. Because the training is driven by loss on the hole training set, there is incentive to focus more on the larger classes. The quality on the smaller classes might suffer from this. This could be a reason to pay attention to balancing the training set with respect to the number of label representatives.

Label 1 is underrepresented in the training set. Its high score on precision attracts attention. A high precision means that it is hardly ever wrongfully identified. Because the label 1 class is so small it makes sense not to predict this label when there is uncertainty. The change of being incorrect is higher because there are not many 'Disgust' faces. The training process makes use of this property.

The conclusion after checking recall and precision is that attention should be paid to balancing the training set with respect to the number of label representatives.

label	0	1	2	3	4	5	6
recall	0.507	0.400	0.371	0.869	0.464	0.737	0.618
precision	0.515	0.846	0.545	0.751	0.466	0.737	0.556
f1	0.511	0.543	0.441	0.805	0.465	0.737	0.585

Table 5: Quality metrics model 6.

Other possibilities for augmentation were not explored. It would be beneficial to know a little more on the training set. Could it already contain augmented data? There is the risk of polluting training set quality by adding more of the same data.

6 Model analysis

We constructed our Neural Network models using the torch.nn package. A neural network consists of several layers and a method (forward - input) that returns the output. A feed-forward network takes the input, feeds it through several layers one after the other, and finally gives the output. In our architectures we choose to include Convolution Layers and Fully connected Layers.

A Convolution layer consists of the modules:

- nn.Conv2D
- nn.BatchNorm2d
- nn.ReLU
- nn.Dropout2d
- nn.MaxPool2d

Also, worth mentioning that the above modules added to a convolution filter sequential, i.e. modules will be added to the level in the order they pass.

For every convolutional layer, we take into account the size of the input and the output channels, which are the number of channels in the input image and those produced by the convolution respectively. Also, we can adjust the size of the convolving kernel (kernel_size), stride the convolution and add padding to both sides of the input. Following, we can choose the operations after applying each convolution layer, such as batch normalization (BatchNorm2d), and apply rectified linear unit

function via ReLU. Furthermore, each channel will be zeroed out independently on every forward call with probability p using samples from a Bernoulli distribution. More specifically, if the adjusted pixels within feature maps are strongly correlated -as is usual at the first convolution layers- then the dropout will not regularize the activations and will otherwise just result in an effective learning rate decrease.

A Fully connected Layer consists of the modules:

- nn.Linear
- nn.BatchNorm1d
- nn.Dropout
- nn.ReLU

Instead of applying a convolution over input as before (Conv2D), we apply a linear transformation to the incoming data. Batch Normalization and ReLU will be helpful for more accurate output.

After evaluating the models as mentioned in the previous sections, we decided to select the Model 5 architecture for our neural network. This model architecture consists of 3 convolutional and 2 fully connected layers. A major difference with previous architectures is the higher dropout probability $p = 0, 4$, which results in a more effective learning rate decrease.

Recall	0.484	0.363	0.365	0.860	0.503	0.721	0.629
Precision	0.561	0.769	0.546	0.750	0.458	0.757	0.539
F1 - Score	0.520	0.493	0.438	0.801	0.479	0.738	0.581

Table 6: Quality metrics Model 5.

The provided table demonstrates more quality measures for the model 5. Where Recall attempts to answer what proportion of actual positives was identified correctly and Precision attempts to answer what proportion was actually correct. Finally, we calculate the f1 score that measures the preciseness and robustness of our model.

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

In order to get an idea on what the layers are doing, the model is fed an example image for each emotion from the testset. The layer values are displayed in Appendix B. The architecture has 3 convolution layers with 64 channels each. Because of max-pooling with a kernel size of 2 the dimensions of the image is decreasing with a factor 2 with each layer. Starting with an image of 48x48, to 24x24, 12x12, 6x6. It can be seen that images with different emotions light up at different channels in different areas. This is the level of activation of the nodes.

References

FER2013 dataset. (n.d.). https://www.kaggle.com/ashishpatel26/facial-expression-recognitionferchallenge. (Accessed: 2021-05-16)

FER2013 dataset discussion. (n.d.). https://www.kaggle.com/ashishpatel26/facial-expression-recognitionferchallenge/discussion/173941. (Accessed: 2021-05-24)

A Appendix - Architectures

```
Development environment: Python 3.8 is used for coding, Anaconda 3 distribution.
The experiments were run on Windows laptops.
Python packages: sys, scipy.io, cv2, numpy, matplotlib, torch
Selected Framework for this assignment: Pytorch


# Description: Code for Assignment 2 - CNNs
# Course:      Computer Vision
# Authors:     Dimitrios Gagatsis
#              Sacha L. Sindorf
# Date:        2021-05-16

import numpy as np

import torch
import torch.nn as nn


# Small architecture to set up training structure
class ConvNet0(nn.Module):
    def __init__(self):
        super(ConvNet0, self).__init__()

        self.layer1 = nn.Sequential(
            nn.Conv2d(1, 8, kernel_size=4, stride=2, padding=1),
            nn.ReLU(True)
        )
        self.layer2 = nn.Sequential(
            nn.Conv2d(8, 12, kernel_size=3, padding=1),
            nn.ReLU(True)
        )
        self.layer3 = nn.Sequential(
            nn.Conv2d(12, 16, kernel_size=4, stride=2, padding=1),
            nn.ReLU(True)
        )
        self.fc = nn.Linear(16*12*12, 7)

        self.sm = nn.Softmax(dim=1)

    def forward(self, x):
        out = self.layer1(x)
        out = self.layer2(out)
        out = self.layer3(out)
        out = out.reshape(out.size(0), -1)
        out = self.fc(out)
        out = self.sm(out)
        return out
```

```
# first trial
class ConvNet1(nn.Module):
    def __init__(self):
        super(ConvNet1, self).__init__()

        self.layer1 = nn.Sequential(
            nn.Conv2d(1, 64, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(num_features=64),
            nn.ReLU(),
            nn.Dropout2d(p=0.1),
            nn.MaxPool2d(kernel_size=2, stride=2),
        )
        self.layer2 = nn.Sequential(
            nn.Conv2d(64, 128, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(num_features=128),
            nn.ReLU(),
            nn.Dropout2d(p=0.1),
            nn.MaxPool2d(kernel_size=2, stride=2),
        )
        self.layer3 = nn.Sequential(
            nn.Conv2d(128, 512, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(num_features=512),
            nn.ReLU(),
            nn.Dropout2d(p=0.1),
            nn.MaxPool2d(kernel_size=2, stride=2),
        )

        self.fc1 = nn.Sequential(
            nn.Linear(6 * 6 * 512, 1024),
            nn.BatchNorm1d(num_features=1024),
            nn.Dropout(p=0.1),
            nn.ReLU()
        )
        self.fc2 = nn.Sequential(
            nn.Linear(1024, 7),
            nn.BatchNorm1d(num_features=7),
            nn.Dropout(p=0.1),
            nn.ReLU()
        )
        self.sm = nn.Softmax(dim=1)

    def forward(self, x):
        out = self.layer1(x)
        out = self.layer2(out)
        out = self.layer3(out)

        out = out.reshape(out.size(0), -1)
        out = self.fc1(out)
        out = self.fc2(out)
        out = self.sm(out)
        return out
```

```
# lower dimensions conv layers
class ConvNet2(nn.Module):
    def __init__(self):
        super(ConvNet2, self).__init__()

        self.layer1 = nn.Sequential(
            nn.Conv2d(1, 64, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(num_features=64),
            nn.ReLU(),
            nn.Dropout2d(p=0.1),
            nn.MaxPool2d(kernel_size=2, stride=2),
        )
        self.layer2 = nn.Sequential(
            nn.Conv2d(64, 64, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(num_features=64),
            nn.ReLU(),
            nn.Dropout2d(p=0.1),
            nn.MaxPool2d(kernel_size=2, stride=2),
        )
        self.layer3 = nn.Sequential(
            nn.Conv2d(64, 64, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(num_features=64),
            nn.ReLU(),
            nn.Dropout2d(p=0.1),
            nn.MaxPool2d(kernel_size=2, stride=2),
        )

        self.fc1 = nn.Sequential(
            nn.Linear(6 * 6 * 64, 1024),
            nn.BatchNorm1d(num_features=1024),
            nn.Dropout(p=0.1),
            nn.ReLU()
        )
        self.fc2 = nn.Sequential(
            nn.Linear(1024, 7),
            nn.BatchNorm1d(num_features=7),
            nn.Dropout(p=0.1),
            nn.ReLU()
        )
        self.sm = nn.Softmax(dim=1)

    def forward(self, x):
        out = self.layer1(x)
        out = self.layer2(out)
        out = self.layer3(out)

        out = out.reshape(out.size(0), -1)
        out = self.fc1(out)
        out = self.fc2(out)
        out = self.sm(out)
        return out
```

```
# lower dimensions fc layers
class ConvNet3(nn.Module):
    def __init__(self):
        super(ConvNet3, self).__init__()

        self.layer1 = nn.Sequential(
            nn.Conv2d(1, 64, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(num_features=64),
            nn.ReLU(),
            nn.Dropout2d(p=0.1),
            nn.MaxPool2d(kernel_size=2, stride=2),
        )
        self.layer2 = nn.Sequential(
            nn.Conv2d(64, 64, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(num_features=64),
            nn.ReLU(),
            nn.Dropout2d(p=0.1),
            nn.MaxPool2d(kernel_size=2, stride=2),
        )
        self.layer3 = nn.Sequential(
            nn.Conv2d(64, 64, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(num_features=64),
            nn.ReLU(),
            nn.Dropout2d(p=0.1),
            nn.MaxPool2d(kernel_size=2, stride=2),
        )

        self.fc1 = nn.Sequential(
            nn.Linear(6 * 6 * 64, 512),
            nn.BatchNorm1d(num_features=512),
            nn.Dropout(p=0.1),
            nn.ReLU()
        )
        self.fc2 = nn.Sequential(
            nn.Linear(512, 7),
            nn.BatchNorm1d(num_features=7),
            nn.Dropout(p=0.1),
            nn.ReLU()
        )
        self.sm = nn.Softmax(dim=1)

    def forward(self, x):
        out = self.layer1(x)
        out = self.layer2(out)
        out = self.layer3(out)

        out = out.reshape(out.size(0), -1)
        out = self.fc1(out)
        out = self.fc2(out)
        out = self.sm(out)
        return out
```

```
# lower dimensions conv layers
class ConvNet4(nn.Module):
    def __init__(self):
        super(ConvNet4, self).__init__()

        self.layer1 = nn.Sequential(
            nn.Conv2d(1, 32, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(num_features=32),
            nn.ReLU(),
            nn.Dropout2d(p=0.1),
            nn.MaxPool2d(kernel_size=2, stride=2),
        )
        self.layer2 = nn.Sequential(
            nn.Conv2d(32, 32, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(num_features=32),
            nn.ReLU(),
            nn.Dropout2d(p=0.1),
            nn.MaxPool2d(kernel_size=2, stride=2),
        )
        self.layer3 = nn.Sequential(
            nn.Conv2d(32, 32, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(num_features=32),
            nn.ReLU(),
            nn.Dropout2d(p=0.1),
            nn.MaxPool2d(kernel_size=2, stride=2),
        )

        self.fc1 = nn.Sequential(
            nn.Linear(6 * 6 * 32, 512),
            nn.BatchNorm1d(num_features=512),
            nn.Dropout(p=0.1),
            nn.ReLU()
        )
        self.fc2 = nn.Sequential(
            nn.Linear(512, 7),
            nn.BatchNorm1d(num_features=7),
            nn.Dropout(p=0.1),
            nn.ReLU()
        )
        self.sm = nn.Softmax(dim=1)

    def forward(self, x):
        out = self.layer1(x)
        out = self.layer2(out)
        out = self.layer3(out)

        out = out.reshape(out.size(0), -1)
        out = self.fc1(out)
        out = self.fc2(out)
        out = self.sm(out)
        return out
```

```
# back to ConvNet3 - higher dropout
class ConvNet5(nn.Module):
    def __init__(self):
        super(ConvNet5, self).__init__()

        self.layer1 = nn.Sequential(
            nn.Conv2d(1, 64, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(num_features=64),
            nn.ReLU(),
            nn.Dropout2d(p=0.4),
            nn.MaxPool2d(kernel_size=2, stride=2),
        )
        self.layer2 = nn.Sequential(
            nn.Conv2d(64, 64, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(num_features=64),
            nn.ReLU(),
            nn.Dropout2d(p=0.4),
            nn.MaxPool2d(kernel_size=2, stride=2),
        )
        self.layer3 = nn.Sequential(
            nn.Conv2d(64, 64, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(num_features=64),
            nn.ReLU(),
            nn.Dropout2d(p=0.4),
            nn.MaxPool2d(kernel_size=2, stride=2),
        )

        self.fc1 = nn.Sequential(
            nn.Linear(6 * 6 * 64, 512),
            nn.BatchNorm1d(num_features=512),
            nn.Dropout(p=0.4),
            nn.ReLU()
        )
        self.fc2 = nn.Sequential(
            nn.Linear(512, 7),
            nn.BatchNorm1d(num_features=7),
            nn.Dropout(p=0.4),
            nn.ReLU()
        )
        self.sm = nn.Softmax(dim=1)

    def forward(self, x):
        out = self.layer1(x)
        out = self.layer2(out)
        out = self.layer3(out)

        out = out.reshape(out.size(0), -1)
        out = self.fc1(out)
        out = self.fc2(out)
        out = self.sm(out)
        return out
```

B Appendix - Feature maps

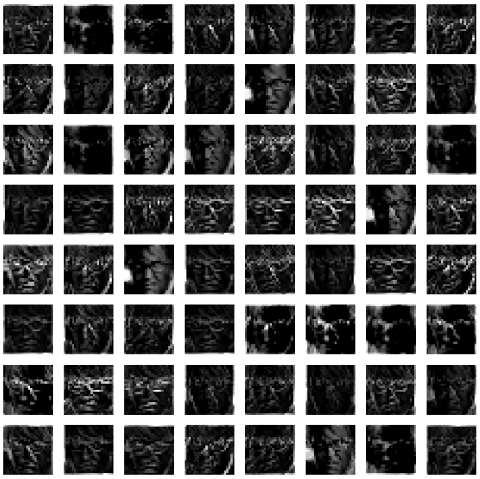

Figure 7: Feature map CONV layer 1, emotion = 0

Figure 8: Feature map CONV layer 2, emotion = 0

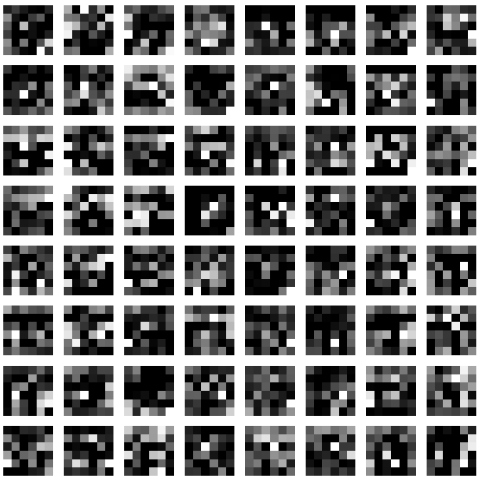

Figure 9: Feature map CONV layer 3, emotion = 0

Figure 10: Feature map CONV layer 1, emotion = 1

Figure 11: Feature map CONV layer 2, emotion = 1

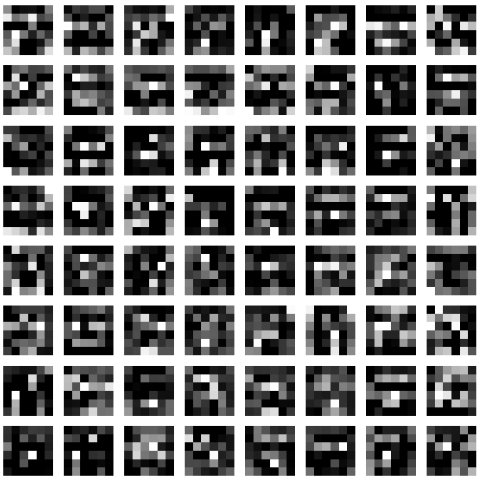

Figure 12: Feature map CONV layer 3, emotion = 1

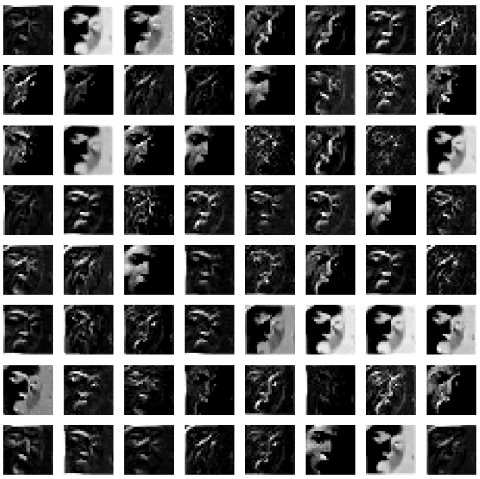

Figure 13: Feature map CONV layer 1, emotion = 2

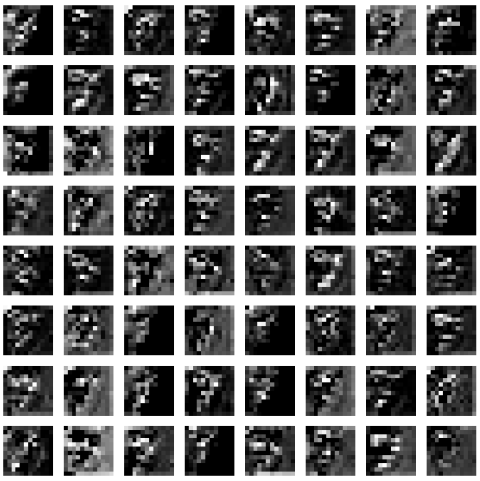

Figure 14: Feature map CONV layer 2, emotion = 2

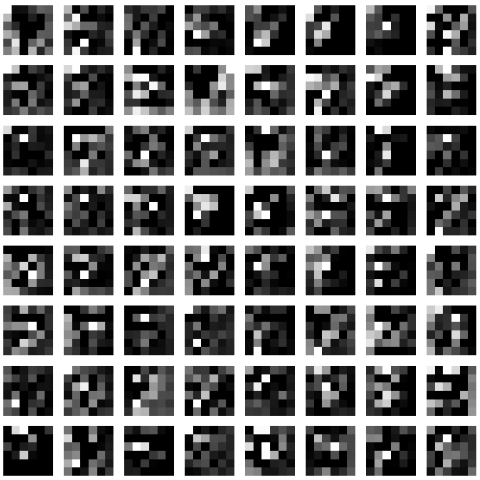

Figure 15: Feature map CONV layer 3, emotion = 2

Figure 16: Feature map CONV layer 1, emotion = 3

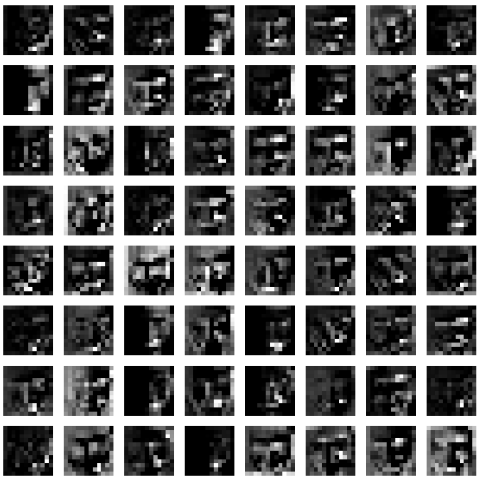

Figure 17: Feature map CONV layer 2, emotion = 3

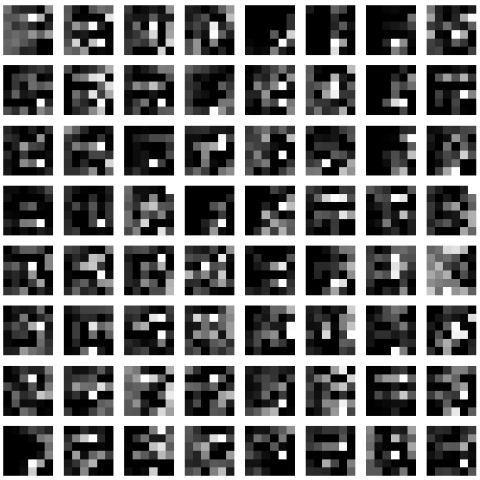

Figure 18: Feature map CONV layer 3, emotion = 3

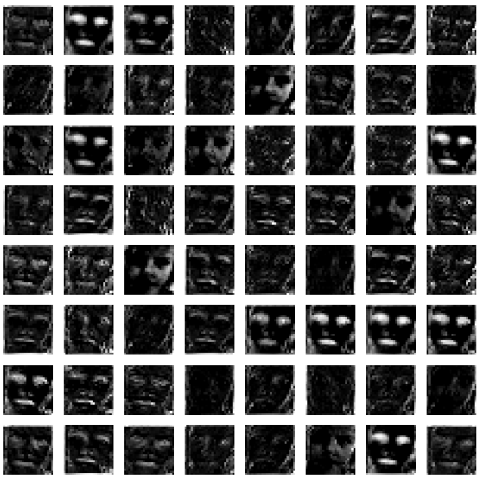

Figure 19: Feature map CONV layer 1, emotion = 4

Figure 20: Feature map CONV layer 2, emotion = 4

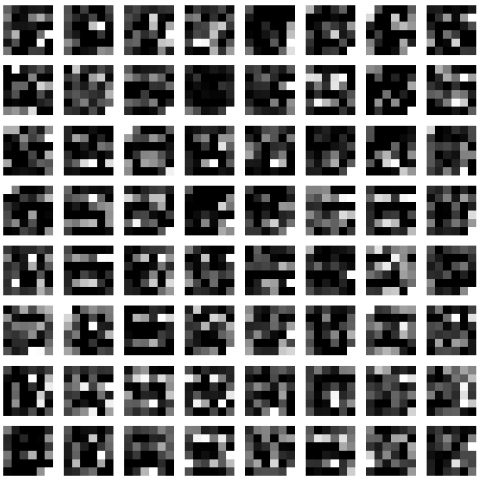

Figure 21: Feature map CONV layer 3, emotion = 4

Figure 22: Feature map CONV layer 1, emotion = 5

Figure 23: Feature map CONV layer 2, emotion = 5

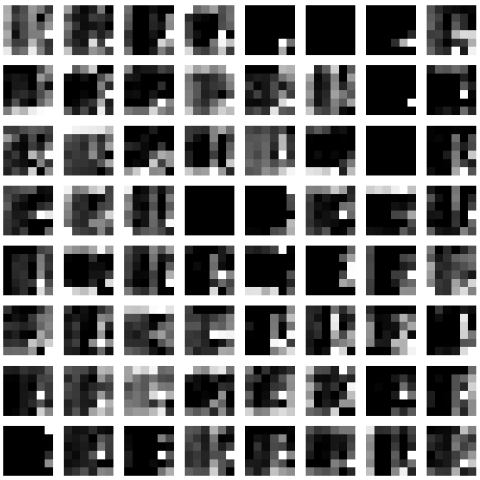

Figure 24: Feature map CONV layer 3, emotion = 5

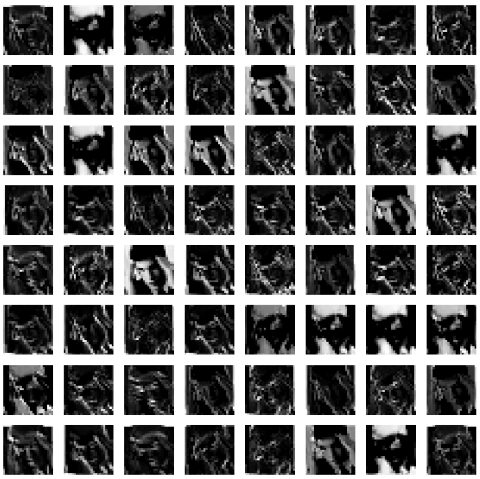

Figure 25: Feature map CONV layer 1, emotion = 6

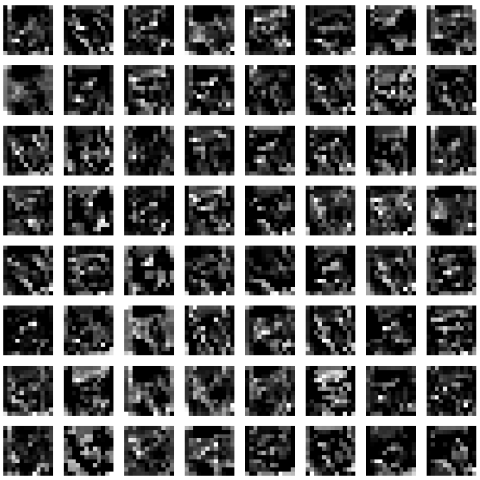

Figure 26: Feature map CONV layer 2, emotion = 6

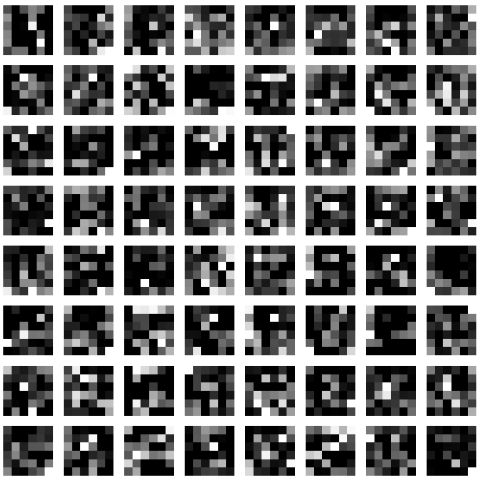

Figure 27: Feature map CONV layer 3, emotion = 6