UNIVERSITETET I OSLO

University of Oslo
Department of Physics

FYS-STK4155 - Applied Data Analysis and Machine Learning

Project 3:

Facial emotion recognition with gradient boosted CNNs

Authors:

Alessia Sanfelici, Federico Santona, Carmen Ditscheid and Andreas Alstad

February 22, 2024

Contents

Abstract

This report investigates the integration of Convolutional Neural Networks (CNNs) and XGBoost for the purpose of emotion detection in images. Employing the inherent strengths of CNNs, we constructed a deep learning model aimed at extracting pertinent features from images rich in emotional content. Further enhancing our approach, we incorporated XGBoost, a potent gradient boosting algorithm, to refine the overall performance of the emotion detection system. Experimental results showcase the efficiency of the combined CNN and XGBoost framework, shedding light on the synergistic potential of deep learning and traditional machine learning techniques for advancing emotion detection applications in visual data. By investigating and tuning several parameters and hyperparameters of the CNN model and that of the XGBoost model, our multiclassification model attained an accuracy score of 56.9% on the validation set.

1 Introduction

After introducing Feed-Forward Neural Networks in *Project 2* to perform classification tasks, we now introduce Convolutional Neural Networks specifically for the emotion recognition of images. Using a picture's property, that its neighbored pixels are related we are able to abstract fully connected layers to convolutional layers. Meaning, we only take the regional behavior of an image in to account when optimizing a parameter, leading to a drastically reduced computational cost. Just which image classification demands.
In the second chapter we motivate and introduce the different layers used in our CNN and thus reason the general architecture of a CNN. In the following chapter we then derive our chosen architecture, reasoning in the properties explained in the previous chapter. We discuss the data, including its prepocessing, and argue the implementation of data augmention to deal with one underrepresented class. Furthermore we treat parameter optimization and the application of the Gradient Bossting library *XGBoost*, and evaluate its applicability for image classification.
The code used to generate these results is available at our GitHub repository. [1]

2 Theory

For image classification convolutional neural networks (CNN) have proven to be the golden standard. This can be attributed to the way that a CNN processes data. Convolution, pooling, subsampling, dropout and padding are common tasks that different layers in a CNN can perform. How they work is explained in the following subsections. These layers can be arranged in a nearly infinite number of combinations and configurations called the *architecture* of the CNN. Because of this flexibility, any CNN can be designed and tuned for practically any task.

2.1 Convolutional neural networks

The backbone of a CNN lies in the name, convolution. In mathematics, convolution is an operation on two different functions that explains how these functions alter each others shape. It can be thought of as one of the functions gliding across the other and summing up the overlaps between the functions at each gliding step. Mathematically, convolution is defined as

$$\begin{aligned}(f * g)(x) &\coloneqq \int_{-\infty}^{\infty} f(t)g(x-t)dt \\ &\equiv \int_{-\infty}^{\infty} f(x-t)g(t)dt\end{aligned} \tag{1}$$

where f and g are arbitrary functions of x, and t is a dummy variable that you integrate over to find the overlap at the moment x. The equivalence in eq. 1 denotes that this is a commutative operation.

Since a CNN commonly deals with images of an integer number of pixels, the two-dimensional discretized version of eq. 1 simpli-

fies to:

$$(f * g)[i, j] = \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} f[m, n] g[i-m, j-n] \quad (2)$$

where m and n are the rows and columns of pixels of a matrix along its height M and width N, respectively. The use of square brackets in eq. 2 around the variables is a notation convention for discrete signals, in contrast to the use of parentheses for continuous signals.

Figures 1 and 2 illustrate how matrix convolution is performed with different strides. The input feature map f is a 5×5 matrix (bright blue) and the *kernel* g (dark blue) is a 3×3 matrix. As visualized in fig. 1 the kernel moves, or *strides*, one pixel at the time along the columns and rows, performing elementwise multiplication of f with g, thus performing the discrete convolution described in eq. 2. This creates the new matrix, visualized in green.

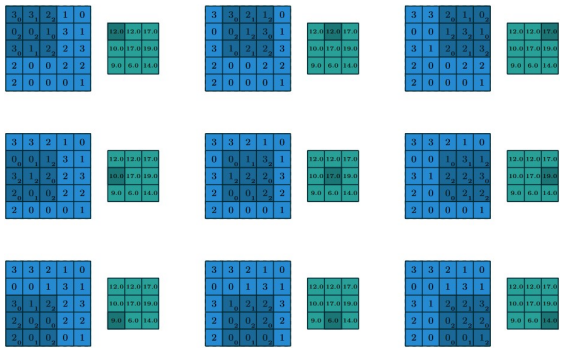


Figure 1: Convolution of a 5×5 input matrix with a 3×3 kernel resulting in a new 3×3 matrix. Illustration aquired from [2].

Note how the kernel has to start at position $[1, 1]$ on f in fig. 1 to fit. By concatinating zeros around the boundary of f, as seen in fig. 2, g is now able to stride two steps at a time in both directions, starting from position $[0, 0]$ on f. This is called *padding*. Different strides result in different matrices with different shapes, depending on the stride in each direction. A symmetric stride can be given as an integer, which means an equal stride in both directions. If the stride is initialized as a tuple of different integers, like $(1, 2)$, and the same padding is used, then the resulting matrix in fig. 2 would have a shape of 5×3.

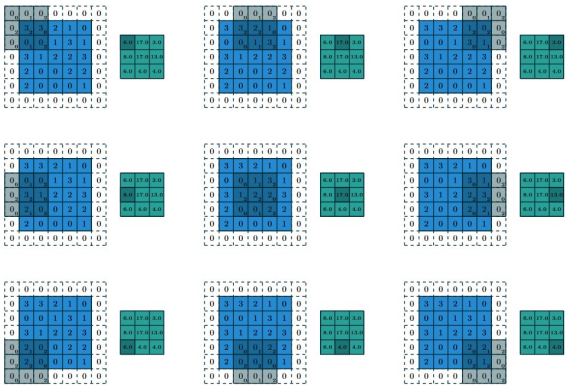


Figure 2: Convolution of a 5×5 input matrix with a 3×3 kernel striding 2 pixels resulting in a new 3×3 matrix. Illustration aquired from [2].

Padding thus allows us to keep the same resolution of an image, if we wish so, but how padding is applied depends on the shape of the kernel. The number of additional rows r and columns c are then defined as:

$$r = \lfloor \frac{\text{kernel height}}{2} \rfloor \cdot 2 \quad (3a)$$

$$c = \lfloor \frac{\text{kernel width}}{2} \rfloor \cdot 2 \quad (3b)$$

where we always round down (note the flooring brackets) to the closest integer of half the kernel shape, and the multiplication at the end ensures padding on both edges of the image.

2.2 Layer types & CNN architecture

Due to all the different ways each unique layer can be initialized and placed sequentially, finding a suitable architecture for a given problem can be hard. These layers are effectively the same as the hidden layers in a Feed Forward NN (FFNN), exept that they not only vary in dimensions, weights and biases, but as well on what tasks they perform. Understanding how they complement (or potentially disrupt) each other is essential to designing an efficient CNN.

2.2.1 Convolutional layer

The first layer initialized in a CNN is typically the convolutional layer. As described in 2.1 these layers apply two-dimentional convolution

using a *kernel*, known as well as *filter*, to extract some feature in the image. These two terms are used interchangeably. Note that any convolutional layer can apply an arbitrary number of kernels. Unless padding is used, these convolutional layers always result in an output with a decreased dimensionality. Padding tends to increase the computational cost of convolution.

2.2.2 Pooling layer

The pooling layer is a crucial element in CNNs as it helps reduce the size of data, making it more manageable. This downsampling operation can take place either before, after, or between convolutional layers in the network. A common practice is to insert pooling layers between convolutional layers. This not only reduces the number of parameters and computational load in the network but also aids in controlling overfitting, contributing to a more robust and generalized model.
In CNNs, two primary types of pooling layers are commonly employed: max pooling and average pooling, as shown in fig. 3.

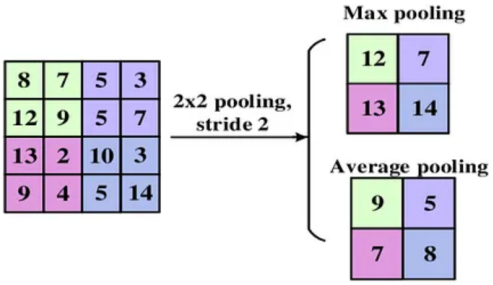


Figure 3: Example of max pooling and average pooling. Illustration aquired from [3].

Max pooling preserves the highest value within designated regions of the input data, discarding all other values. This approach allows for the extraction of the most crucial features in each region while simultaneously eliminating less pertinent information. Conversely, average pooling calculates the average value of the elements within each region. Although this method is less likely to emphasize specific features, it provides a smoother downsampling of the input.

2.2.3 Flattening layer

As suggested by its name, a flattening layer simply converts the image matrix (or tensor) to a one-dimensional array, as seen in fig. 4. This is often necessary when transitioning from the convolutional part of a CNN to a fully connected part that connects to the output layer.

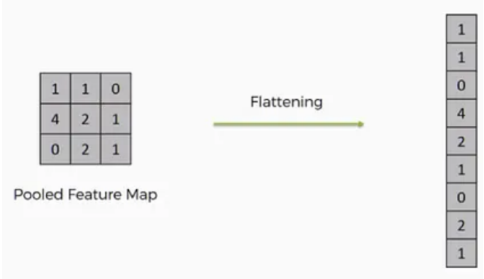


Figure 4: Example of the operation of a flattening layer. Illustration aquired from [3].

2.2.4 Fully connected layer

Just like in ordinary NNs, the fully connected, or *dense*, layer contains the neurons that learn to classify from the extracted features produced by previous layers. Fully connected layers are often used in the latter part of a CNN architecture, following convolutional and pooling layers. Their weights and biases are updated via backpropagation and gradient descent.

2.2.5 Dropout layer

To combat overfitting it is common to include one or multiple *dropout* layers. The purpose of dropout is to prevent co-adaptation of neurons. Dropout means that a random fraction of neurons are ignored during each epoch, forcing the rest of the neurons to compensate. This aids in the generalizability of the model as the neurons that are not ignored have to be trained and get updated with a 'new perspective' each time. When dropout is applied, the network becomes more robust and is less likely to rely heavily on specific neurons. This, in turn, helps improve generalization to new, unseen data.

2.2.6 Output layer

The output layer is another fully connected layer with the number of neurons equal to the number of classes. It is the final layer in a neural network, and its design depends on the specific task the network is intended to perform. The output layer in a CNN is responsible for producing the final predictions based on the hierarchical features extracted by the convolutional and pooling layers. For image classification tasks with C classes, the output layer typically consists of C neurons. The activation function is often softmax, converting the network's raw output into probability scores for each class.

2.3 Gradient boosting with XGBoost

XGBoost, an abbreviation of "Extreme Gradient Boosting", is a gradient-boosting method for decision trees and a highly optimized independent library used to increase the generalizability and predictive power of a model. It is based on an ensemble model known as the decision tree and the "classification and regression trees" (CART) algorithm. An illustrative example of a CART is shown in fig. 5. [4]. The CART algorithm divides the data set at a *root node* into two subsets based on features and corresponding thresholds and conditions, and then divides those data subsets at *interior nodes* based on new thresholds and so on, until a specified *max depth* is reached. The last nodes are called *leaf nodes* where the final classification score of a single instance (like an image) is calculated. Each node is connected by branches which represent the decisions themselves. Unlike with random forests, where trees are created in parallel, XGBoost creates trees sequentially. The first tree used is called a *weak classifier* because it performs slightly better than a random guess. The next tree tries to correct the mistakes of the previous tree by fitting its residuals and predicting its errors and so on, until the combination of weak classifiers results in a solid classifier and the cost function is minimized. The gradient boosting part comes in when each new tree tries to iterate in the direction of the steepest reduction of errors, equivalent to gradient descent. Ensemble methods like gradient boosting are useful when the dataset used for training a model is unbalanced. To combat overfitting each tree is rather shallow with few interior and leaf nodes. This accumulation of error fitting allows for high predictive power without overfitting. The tree accumulation can be described as

$$\hat{y}_i^{(t)} = \sum_{k=1}^{t} f_k(x_i) = \hat{y}_i^{(t-1)} + f_t(x_i) \qquad (4)$$

where each new prediction $\hat{y}_i^{(t)}$ of some data x_i is a sum of the previous predictions and the newest tree f_t. The definition of a tree f_t is described in the XGBoots documentation as

$$f_t(x) = w_{q(x)}, w \in \mathbf{R}^T, q : \mathbf{R}^d \to 1, 2, ..., T \quad (5)$$

where a leaf-score-vector w gets its data point d assigned by a function q. T is the number of leaves. [4]

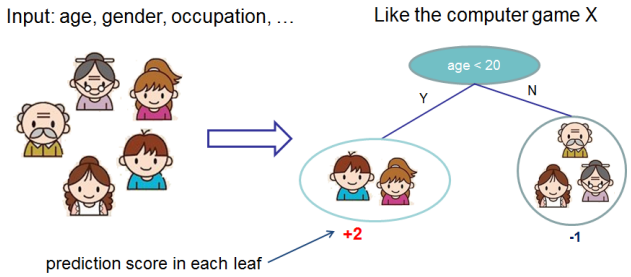


Figure 5: Illustration of the general principle of a CART. The features of the people are used to separate the people according to a threshold and get a prediction score at the leaf that the people are placed into. In this case, age is a good predictor of video game habits. Aquired from [4].

XGBoost further avoids overfitting by adding a regularization term to the cost function, resulting in the *objective function* that will be optimized, which is given in eq. 6 as:

$$\begin{aligned} \text{obj}(\theta) &= L(\theta) + \Omega(\theta) \qquad (6) \\ &= \sum_{i}^{n} l(y_i, \hat{y}_i^{(t)}) + \sum_{i=1}^{t} \omega(f_i) \end{aligned}$$

where $L(\theta)$ is the cost function, $\Omega(f_k)$ is the regularization term containing the complexity (ω) of each tree (f_i) at each iteration t. y is some data and $\hat{y}$ is the predicted value. XGBoost defines the complexity ω of a tree f as

$$\omega(f) = \gamma T + \frac{1}{2}\lambda \sum_{j=1}^{T} w_j^2 \tag{7}$$

where γ is used for pruning the trees if the reduction of the loss score is not substantial enough, and λ is simply the L^2 regularization parameter. Furthermore, XGBoost uses a second order Taylor expansion along with the tree accumulation in eq. 4 to produce the objective function for the t'th tree:

$$\begin{aligned}\text{obj}^{(t)}(\theta) \approx \sum_{i=1}^{n}[g_i f_t(x_i) + \frac{1}{2}h_i f_t^2(x_i)] \\ + \gamma T + \frac{1}{2}\lambda \sum_{j=1}^{T} w_j^2\end{aligned} \tag{8}$$

where

$$\begin{aligned} g_i &= \partial_{\hat{y}_i^{(t-1)}} l(y_i, \hat{y}_i^{t-1}) \\ h_i &= \partial^2_{\hat{y}_i^{(t-1)}} l(y_i, \hat{y}_i^{t-1}). \end{aligned}$$

Decision tree ensemble methods like XGBoost are useful when working on structured or tabulated data, which is why it might be useful to let a CNN extract the features of the data set and let XGBoost replace the fully connected layers of the CNN, potentially achieving greater generalizability.

3 Method

This project's objective was to employ Convolutional Neural Networks for image classification. The model underwent training and testing using a designated dataset [5], which included 35,685 instances of 48x48 pixel grayscale facial images, of which 4 examples are illustrated in fig. 6. These images were pre-organized into two folders representing the training set (28709 images) and the test set (7178 images). Furthermore, the images were already labeled based on the depicted emotion in the facial expressions, namely: angry, disgusted, fearful, happy, neutral, sad, and surprised.

(a) Disgusted **(b)** Sad **(c)** Surprised **(d)** Happy

Figure 6: Examples of images taken from the data set, together with their emotion label.

Using Keras, which is an application programming interface (API), for the machine learning library Tensorflow we designed a CNN architecture made for image classification with an explanation of the choice of layers and their parameters in section 3.4. The Adam algorithm was used to dynamically adjust the learning rate of the model. The cost function we used for the CNN model was the categorical cross-entropy (CCE) loss function, which is the cross-entropy loss of the softmax function, given in eq. 9 by

$$CCE = -\sum_{i}^{C} t_i \log\left(\frac{e^{S_i}}{\sum_j^C e^{S_i}}\right) \tag{9}$$

where C are all the different classes, t_i is the true label (groundtruth), and S_i is the score for class i.

The grid search was performed on the ML nodes platform for machine learning hosted by the University of Oslo. [6]

3.1 Data preprocessing

We rescaled the pixel values from $[0, 255]$ to $[0, 1]$. Furthermore we encoded the labels in a one-hot array.

3.2 Data balancing and augmentation

The frequency distribution plot depicted in fig. 7 clearly shows that the 'disgusted' class is significantly underrepresented. Indeed, it consists of only 549 images (both training and test set),

a notable difference when compared to the approximately 4000-7000 images present in the other classes.

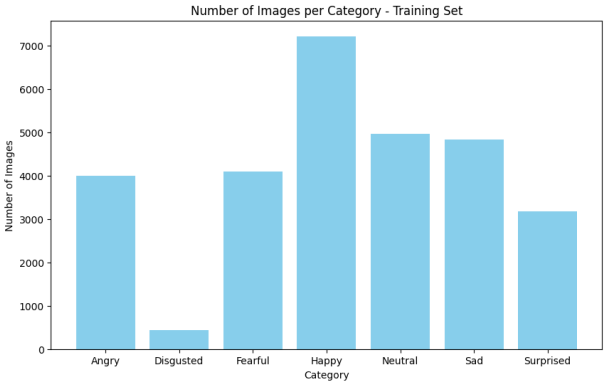


Figure 7: Frequency distribution of the classes in the training set. The 'disgusted' class is significantly underrepresented.

To address the underrepresentation of this particular class, we applied balancing to the training data, ensuring an equal number of images for each class. To stretch the number of samples in the underrepresented class ('disgusted'), we performed data augmentation using Tensorflow's *Image Data Generator* function [7]. The latter function applies rotation, shifting, zooming, flipping and change in brightness to the provided images and thus artificially creates new images, which are used to further train the CNN.

Nevertheless, the introduction of data augmentation led to a decline in the model's performance. Consequently, we opted to abandon data augmentation as well as continue working with the original unbalanced dataset without making any additional modifications. This decision is further elaborated in the discussion section.

3.3 Choosing parameters by grid search

In an effort to discover the optimal set of parameters for the network, we conducted a 3-dimensional grid search. We iterated over the the learning rate, the regularization parameter and the batch size, testing the following parameter values: $\eta = log_{10}([-5, -4, -3, -2, -1])$, $\lambda = log_{10}([-5, -4, -3, -2, -1])$ and the *batch size* $= [8, 16, 32, 64, 128]$.

3.4 Architecture

Selecting the architecture for a CNN is a critical decision that significantly influences the overall performance of the network. The chosen architecture plays a key role in determining how well the CNN can learn and represent patterns in the data, ultimately affecting its ability to successfully accomplish the given task. The initial architecture we settled for was inspired by an article on how one might start training a neural network for emotion recognition using the same dataset we apply in this project. [8]

The number of convolutional layers was the first test, followed by the number of kernels per convolutional and number of neurons in the fully connected layers. Their default settings were as follows: 3 convolutional layers with (32, 64, 64) number of kernels, each with a size of 3×3 pixels, 3 pooling layers with the same size of 2×2 pixels, a dropout rate of 30%, and lastly 3 fully connected layers with (64, 7, 7) neurons. The last fully connected layer was held constant at 7 to stay commensurate to the number of image classes. All layers had a default striding of 1. These default parameters were changed after each test suggesting the optimal value. Due to time constraints every change of these parameters were applied across all layers instead of per layer. For the same reason the number of epochs was limited to 50, even if the performance scores did not always fully converge.

3.5 Gradient Boosting using XGBoost

We attempted to push the performance of our model further by introducing gradient boosting just after feature extraction. XGBoost was used to boost the classification capabilities after the CNN had extracted all the features during convolution. In our work, XGBoost replaced some of the fully connected layers to perform the classification using the flattened feature output of

our model. The output of the different fully connected layers were used as input to the XGBoost model for comparison. The same hyperparameters used for the CNN model were initially also used for XGBoost to guarantee consistency.

The objective function chosen for this project was the "multi:softprop" (multi-class softmax function) objective which results in an output vector of dimensionality datapoints × classes. "mlogloss" (multi-class negative log-likelyhood) was used as the evaluation metric for the objective function with the validation data.

The XGBoost parameters we survey in this work are the maximum depth of the boosted trees, XGBoost's own learning rate, number of boosting rounds, λ in the regularization term of eq. 8, and finally the pruning parameter γ.

4 Results

First, the learning rate, L_2 regularization and batch size values which resulted in the best accuracy and loss scores from a 3D grid search is covered, followed by the tuning of parameters in the different layer related to feature extraction in the CNN model. Then, the accuracy and loss scores of the final CNN model is plotted agains the training epochs. Lastly, how the introduction of XGBoost and the tuning of its hyperparameters altered the CNN model's predictive performance is covered, with a brief demonstration of XGBoost's speed at the end.

The scores for the training data and validation data are plotted as whole and dashed lines, respectively, in most of the figures, and the validation plot labels are abbreviated with "val.".

4.1 Tuning the CNN's parameters and layers

Figs. 8 and 9 show the results from the 3D grid search, where $\eta = 10^{-5}$, $\lambda = 10^{-4}$ and a batch size of 128 achieved the highest accuracy running over 300 epochs.

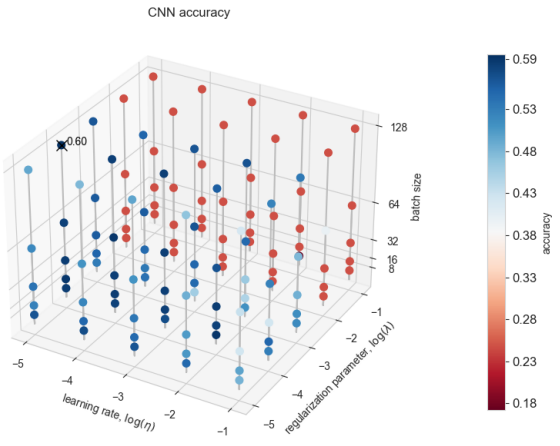


Figure 8: Accuracy score on the test data as a function of learning rate, regularization parameter and batch size: The highest accuracy 60% is yielded by the following combination of parameters: $\lambda = 10^{-4}$, $\eta = 10^{-5}$ and batch size equal to 128.

The values of η and λ seemed rather low, so two quick test runs where conducted to probe the reliability of the grid search; one with the optimal parameters found in fig. 8 and the other with $\eta = 10{-}3$, $\lambda = 10^{-4}$ and batch size of 64 from a previous grid search attempt shown in fig. 30a, to be compared.

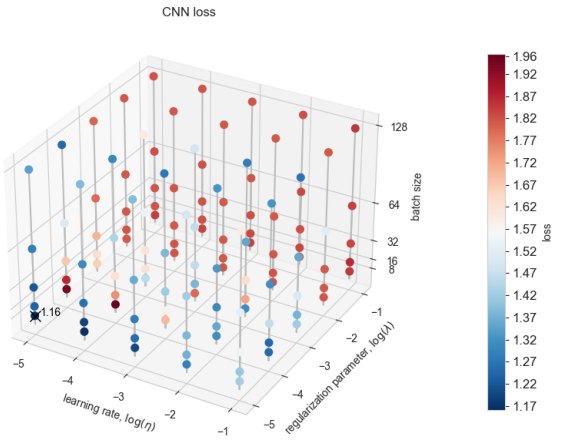


Figure 9: Loss score on the test data as a function of learning rate, regularization parameter and batch size: The lowest loss 1.1608 is yielded by the following combination of parameters: $\lambda = 10^{-5}$, $\eta = 10^{-5}$ and batch size equal to 8.

Running the tests over 50 epochs, the prior achieved an accuracy of 26.71%, while the latter achieved an accuracy of 56.92%, so we chose

the parameters suggested by the results from our previous grid search.

In figs. 10 and 11 we attempted to find a suitable number of convolutional-pooling-layer pair looking at the validation accuracy and loss as function of epochs, respectively. The legend was accidentally misnumbered by 1, so the plot labeled as 2 convolutional layers with dropout was actually 3 layers, which performed the best.

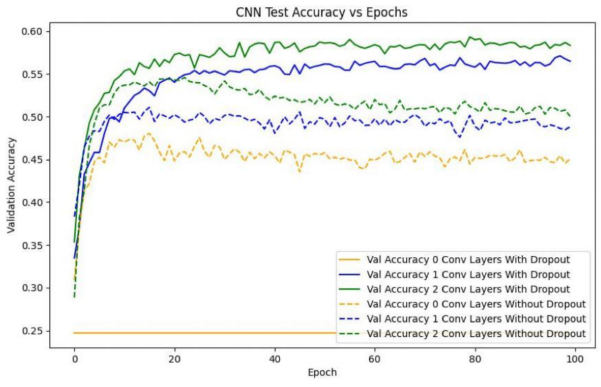


Figure 10: Accuracy score for the CNN using different number of convolutional layers, with and without dropout.

Using only 1 convolutional layer with dropout seems to hinder any possible learning for the model. 3 layers with dropout seems to perform the best, but the steady increase in validation loss regardless of number of layers could be cause for concern, but this behaviour was only encountered once during this project.

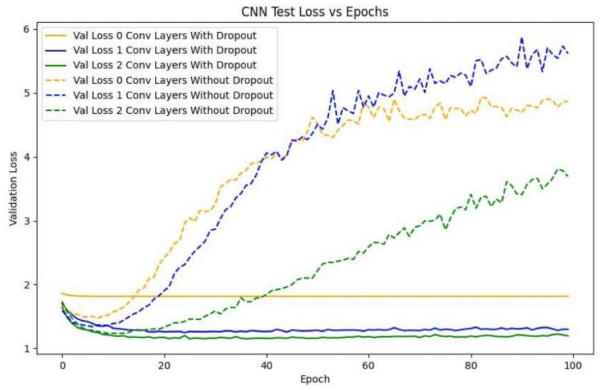


Figure 11: Loss score for the CNN using different number of convolutional layers, with and without dropout.

The score from different numbers of filters per convolutional layer is shown in figs. 12 and 13.

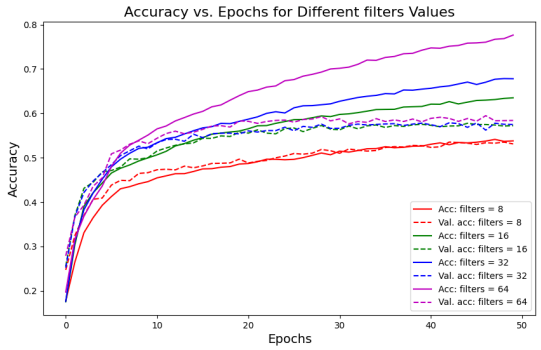


Figure 12: Accuracy scores for the training and validation data of the CNN using different number of filters.

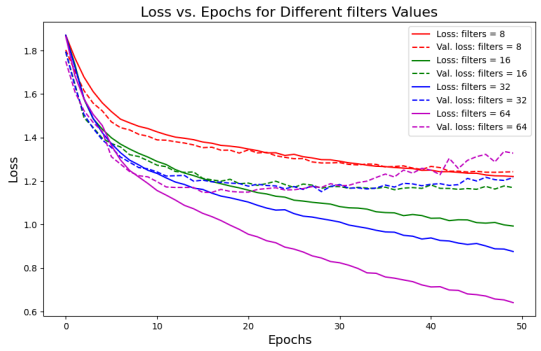


Figure 13: Loss scores for the training and validation data of the CNN using different number of filters.

Here, 3 convolutional-poolng-layer pairs are used, where the first convolutional layer is given the number of filters as shown in the legend of the plots, while the two next convolutional layers given twice as many layers. 16, 32 and 64 filters appears to give negligible gains in accuracy when performed on the validation data, even though the training accuracy steadily increases. Even though the model achieved the lowest loss with 64 filters on the training data, for the validation data it began to rise again after $\sim$ 25 epochs. 16 filters gave a slightly lower loss than 32 for the validation data, but for the training data 32 performed the best by a greater margin. From this we consider 32 filter to be optimal.

In figs. 14 and 15 different sizes of kernels with and without padding using the functions in eq. 3 to accommodate for the larger kernel sizes.

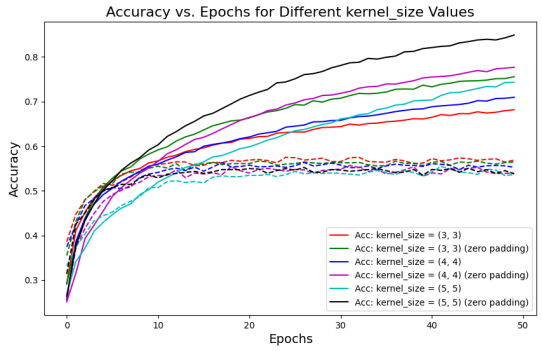


Figure 14: Accuracy scores for the training (whole lines) and validation (dashed lines) data of the CNN using different kernel sizes with and without padding.

Increasing the kernel size and introducing zero-padding offered no noticeable benefits for the model's accuracy, and the loss score suffered considerably. Therefore, we kept the default kernel size of 3×3.

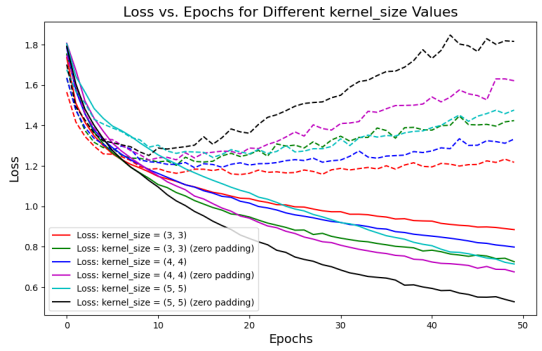


Figure 15: Loss scores for the training (whole lines) and validation (dashed lines) data of the CNN using different kernel sizes with and without padding.

Increasing the dropout rate in figs. 16 and 17 offer no noticeable gains in accuracy. However, there is a slight decrease in loss for the validation data with a dropout rate of 40%, even though that is not the case with the training data. Therefore, 40% dropout is used going forward.

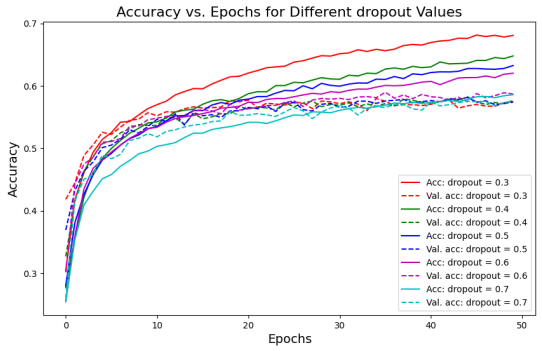


Figure 16: Loss scores for the training (whole lines) and validation (dashed lines) data of the CNN using different kernel sizes with and without padding.

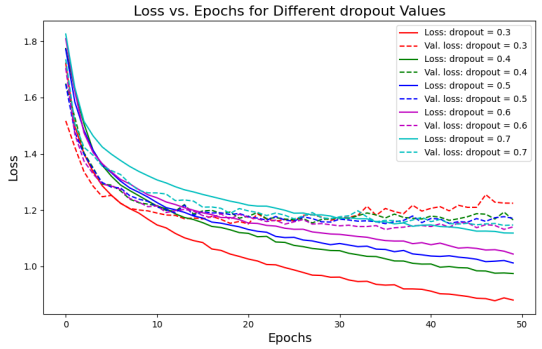


Figure 17: Loss scores for the training (whole lines) and validation (dashed lines) data of the CNN using different kernel sizes with and without padding.

Increasing the size of the max-pooling filter had a slight negative effect on the performance of the model, reducing the accuracy on the validation set the larger the pooling filter was. Even though the loss with a max-pooling filter size of 2×2 surpased after $\sim$ 25 epochs, likely resulting in overfitting, the same size performed better than the others below 25 epochs, so 2×2 was deemed large enough.

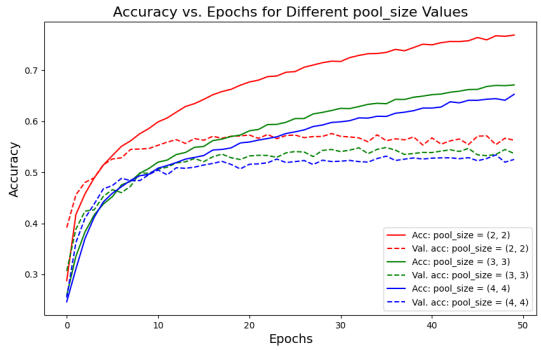


Figure 18: Loss scores for the training and validation data of the CNN using different max-pooling filter sizes.

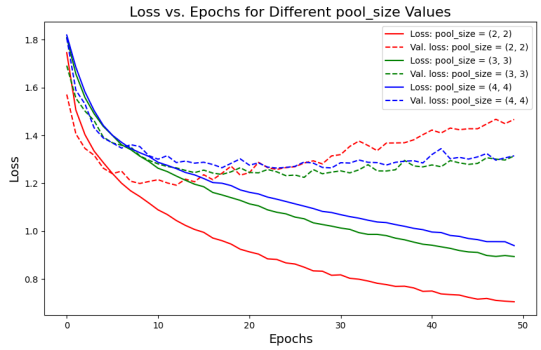


Figure 19: Loss scores for the training and validation data of the CNN using different max-pooling filter sizes.

The number of neurons in the first fully connected layer had a negligible effect on the accuracy with the validation data, as shown in figs. 20 and 21.

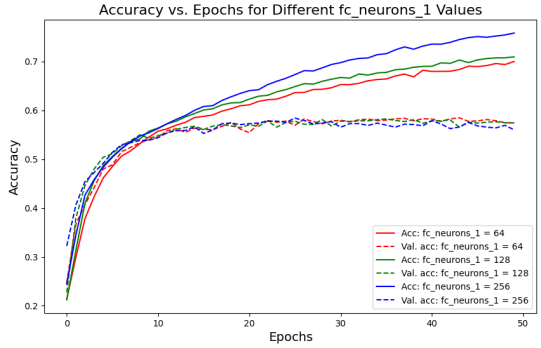


Figure 20: Accuracy scores for the training and validation data of the CNN using different number of neurons in the first fully connected layer.

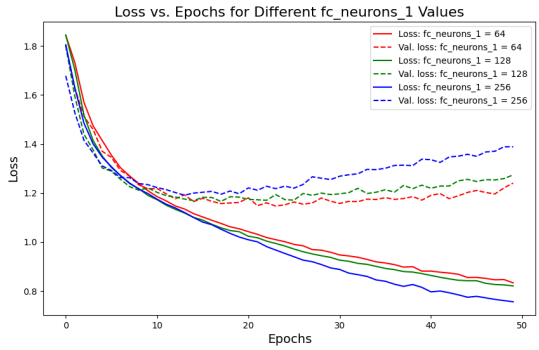


Figure 21: Loss scores for the training and validation data of the CNN using different number of neurons in the first fully connected layer.

However, since the increase in loss across the epochs for the validation data in fig. 21 is compounded by the number of neurons, 64 neurons seems to be sufficient, as 128 and 256 neurons appears to contribute to overfitting.

The same trend regarding the accuracy applies for the number of neurons in the second fully connected layer in figs. 22 and 23, the loss for the validation data barely changes with the amount of neurons. An interesting detail is that 7 neurons performed better on the training data than 32, but almost equal to 128 neurons, and finally worse than 64 neurons. But despite the variantion in training scores, the validation scores indicate that the default of 7 neurons was sufficient to potentially limit unnecessary use of computational power.

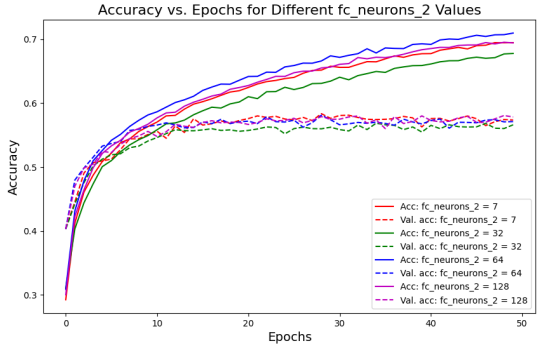


Figure 22: Accuracy scores for the training and validation data of the CNN using different number of neurons in the second fully connected layer.

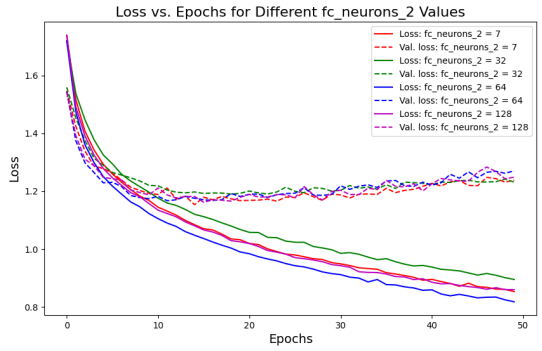


Figure 23: Loss scores for the training and validation data of the CNN using different number of neurons in the second fully connected layer.

Applying these suggested parameters for the different layers our model achieved an average accuracy score of 57% on the validation data, as shown in fig. 24. The accuracy on the training data steadily increases, although at a decaying rate, the validation accuracy appears to have plateaued at around the $25 - 35$ epoch range, after which it seems to slowly decay the longer the training went on.

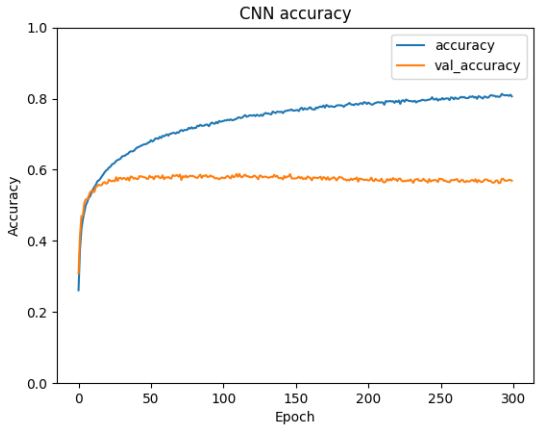


Figure 24: Accuracy scores of the CNN with optimal values of the parameters we investigated.

The problematic trend can also be seen in fig. 25 where after ~ 25 epochs the model starts to get overtrained without learning anything, according to the validation scores. The loss score increases, indicating overfitting, meaning the models ability to generalize suffers.

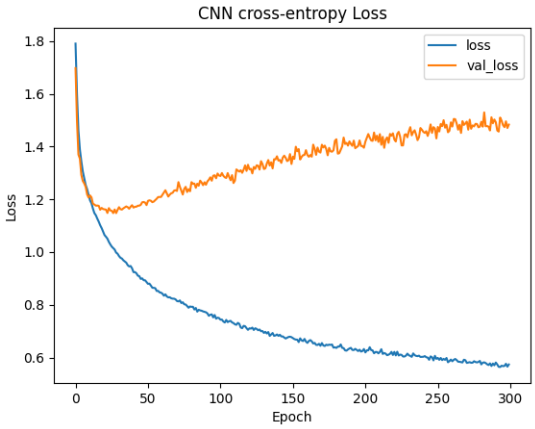


Figure 25: Loss scores of the CNN with optimal values of the parameters we investigated.

4.2 Tuning the XGBoost model

XGBoost was applied at 4 different position in the CNN architecture to probe the optimal placement. The layers XGBoost recieved input from was the *flatten layer*, the *first fully connected layer*, the *second fully connected layer*, and finaly the *output layer*, achieving accuracy scores of 53.52%, 55.20%, 55.03% and 55.13%, respectively. The XGBoost model performs marginally worse than the ordinary CNN model when taking the output of the first fully connected layer as input, as shown in tables 2-6 in the appendix.

Lastly, the speed of XGBoost was compared to that of the ordinary CNN model. As tuning XGBoost's parameters showed no tangible benefits, the default values were used. The CNN model was first trained on the training data over 50 epochs. The XGBoost model was initialized by taking the output of the first fully connected layer as input. Computation time for both of the models was measured calling `time.time()` around the `.predict()` calls for both of the models. This test was run on a desktop PC with an AMD Ryzen 5 3600 6-core processor with 32 GB of 3200 MHz memory (RAM). The CNN's prediction took `0.98 s` and for XGBoost it took an impressive `0.01 s`. No compatible GPU was used to accelerate the processes.

4.3 Performance of the model

We achieved an overall accuracy score of 56.9% on the validation set, with scores per emotion tabulated in table 1, which are also the values along the diagonal of the relative matrix in fig. 27.

Table 1: Accuracy scores across each emotion evaluated from the validation data.

Class	Emotion	Accuracy [%]
0	Angry	47.91
1	Disgusted	41.44
2	Fearful	34.18
3	Happy	79.09
4	Neutral	54.58
5	Sad	46.11
6	Surprised	69.68

Fig. 26 shows a confusion matrix with the true labels of the images in the test data versus their predicted labels. The brightness of the diagonal squares imply how well the model performed. Unsurprisingly, our model found class 1 ('disgusted') to be the most difficult to predict and class 3 ('happy') to be the easiest, yet the reciever operating characteristic (ROC) curve in fig. 28 indicate that our model performs appreciably better than randomly guessing on a 'disgusted' label.

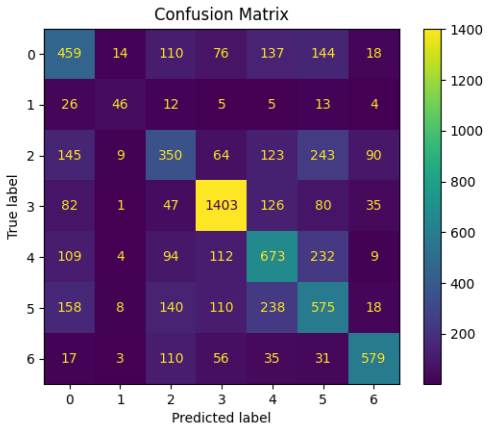


Figure 26: Confusion Matrix with absolute values: The correct classifications, corresponding to the values on the diagonal, significantly dominate the wrong classifications.

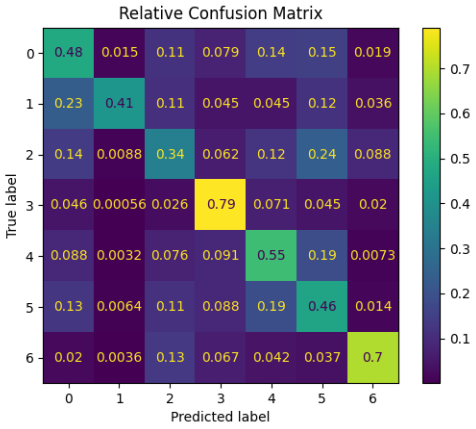


Figure 27: Confusion Matrix: To perform a reasonable analysis despite class imbalance, we normalized the rows by dividing by the cardinality of images per class. We observe an explicitly strong performance on the classes 3 and 6, respectively 'happy' and 'surprised'. While the first is expectable, since the 'happy' class contains the most images, the latter is remarkable since 'surprised' is the class with the second least images. We note that the underrepresented class 2 performs solid and even outperforms the third class 'fearful', validating our analysis in Fig. 28.

The "area under ROC" (AUC) value indicates how well the model is able to distinguish a class from all the other classes.

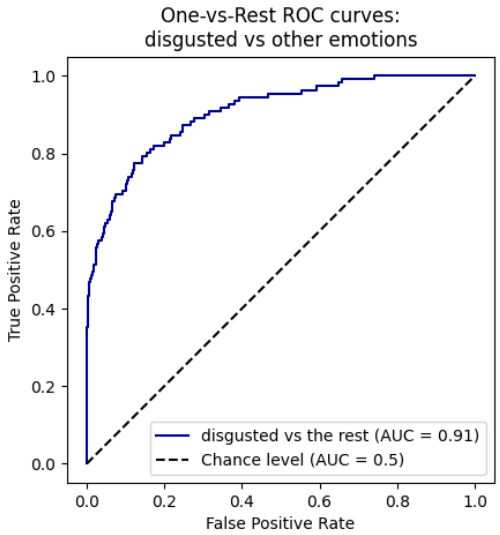


Figure 28: Multiclass ROC (One-vs.-Rest): Our model performs already significantly better on the underrepresented class than the random classifier, thus indicating that one of the previously described cases applies and therefore data balancing and augmentation decrease the model's performance. Furthermore, see Fig. 27.

With an AUC = 1 being the ideal case, values of 0.91 and 0.94 for the 'disgusted' and 'happy' classes, respectively, confirm that our model exhibits a serviceable ability of class seperability.

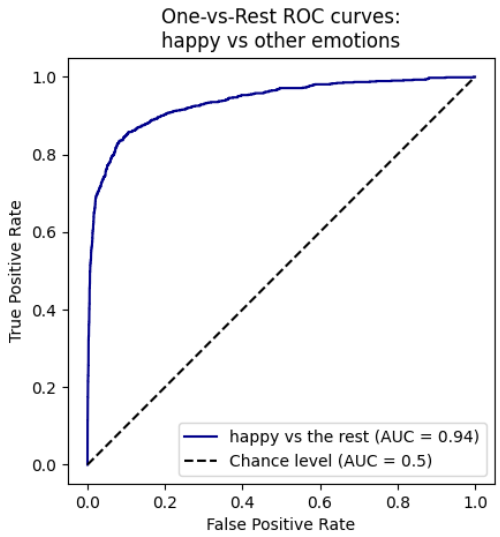


Figure 29: Multiclass ROC (One-vs.-Rest): As predicted, our model performs significantly better on the over class than the random classifier. Furthermore, see Fig. 27.

5 Discussion

While balancing classes to e. g. prevent bias towards the majority classes appears intuitive, it bears a few risks. Firstly, augmenting the underrepresented class emphasizes the possibly in the train data contained anomalies and thus leads to an overfitting on the underrepresented class. Furthermore, by introducing balancing we violate the assumption that train and test data follow the same underlying distribution since the probability of the image being part of the underrepresented class is lower in the latter data set which reflects reality. Lastly, the resampling may be unnecessary because the cardinality of the underrepresented data in the train data set already suffices for a solid classification performance. [9]

The results from the 3D grid search are somewhat unreliable. A previous attempt (see figs. 30a and 30b) indicate that setting the learning rate to 10^{-3} and the L_2 regularization to 10^{-4} with a batch size of 64 would yield the highest accuracy, but not the lowest loss. This could indicate some numerical instability and undermines the reproducibility of our model and its scores. The two test runs using the two sets of parameters suggested by our previous and latest grid searches indicate that the learning rate suggested in fig. 8 was too low, impeding the model's ability to learn properly. It could also be due to incorrectly packing or unpacking the results in a `.npy` (NumPy binary) file. Saving the results this way was necessary in order to adjust the plots in a visually coherent way without rerunning the grid search, which took up to two days on accelerated hardware provided by ML Nodes.

Increasing the number of convolutional layer filters would give the CNN greater depth, which in theory would give the model a greater learning opportunity, but as fig. 12 suggests that is not always the case, and it may also unnecessarily increase computational costs. This decision is made across most of the parameter trials, as the accuracy gains are often not enough to justify the increase in computational costs.

How striding affects the predictive performance of the model was omitted due to challenges related to dimensionality during convolution. For the same reason zero-padding was added to the pooling layer size tests as changing from the default pooling size of 2×2 quickly resulted in TensorFlow throwing "negative dimensionality" errors. The 48×48 size of the images in our dataset constrained the use of layers and parameters which would result in downsampling of the images, like striding and larger pooling filters.

We tested larger number of neurons for the first fully connected layer than the second due to a common heuristic approach of using a funnel-like structure from the first fully connected layers towards the output layer in an attept to limit overfitting. The wider fully connected layer would then be able to learn from the myriad of features after convolution is done and condense the knowledge for the subsequent fully connected layers, further reducing dimensionality and consequently computational costs. How-

ever, since the architecture and parameters of the CNN is highly data- and task-dependent, this heuristic choice does not necessarily carry any benefits in of itself. The dataset itself has a rather limited variation of images; all are of equal size and in grayscale. Had the images contained color and varying resolutions, our CNN might have been able to learn faster and generalize better to unseen data.

As decision tree based methods are regarded as the golden standard for tabulated and highly structured data, we assumed it would pair well with our CNN model as the "classifier part", but XGBoost proved to be rather insensitive to our tuning approach, providing our CNN model negligible gain or loss of predictive performance. For the parameter tuning of the XGBoost model, some parameters like *subsample*, *colsample_bytree* and *min_child_weight* were omitted to save time. Neural networks tend to operate as "black boxes" in that how features are interpreted and extracted is not always very clear. However, XGBoost demonstrated its impressive speed as it was able to make predictions many times faster than the CNN on the same test data. While XGBoost might perform best with regards to accuracy on tabulated data if the number of features is less than the number of data entries, it is performant enough to consider including sometimes regardless of the task.

6 Conclusion

We examined the model's behavior in dependence of multiple parameters such as the learning rate, the regularization parameter and the batch size. Furthermore we observed the model's performance under different architectures by varying the number of convolutional layers and optional dropout. Moreover we tuned the kernel sizes, filters and optional padding to build a more nuanced classification model, but few parameters were ultimately adjusted from the base architecture we started with.
We introduced data balancing and augmentation but discarded it as we noticed a decrease in performance.
While XGBoost was initially introduced to boost the predictive performance of the CNN, it became clear that it could not contribute much, likely due to the non-tabular nature of the extracted features. However, XGBoost demonstrated one of its main advantages, namely its speed. It achieved similar prediction scores to the CNN on the test data, but almost two orders of magnitude faster.
Ultimately, we attained an accuracy of 56.9% on the test data, benchmarked by an accuracy of 72.16% in this scientific paper [8], we consider this a solid performance given the time and resources for this project.

References

[1] Alessia Sanfelici, Federico Santona, Carmen Ditscheid and Andreas Alstad. *Project3*. 2023. URL: https://github.com/FedericoSantona/Project3.

[2] Vincent Dumoulin and Francesco Visin. *A guide to convolution arithmetic for deep learning*. 2018. arXiv: 1603.07285 [stat.ML].

[3] *Medium, Convolutional Neural Networks (CNN) — Architecture Explained*. https://medium.com/@draj0718/convolutional-neural-networks-cnn-architectures-explained-716fb197b243. Accessed: November 30, 2023. June 1, 2022.

[4] Tianqi Chen and Carlos Guestrin. "XGBoost: A Scalable Tree Boosting System". In: *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD '16. San Francisco, California, USA: ACM, 2016, pp. 785–794. ISBN: 978-1-4503-4232-2. DOI: 10.1145/2939672.2939785. URL: http://doi.acm.org/10.1145/2939672.2939785.

[5] *Access to the used data set.* https://www.kaggle.com/datasets/ananthu017/emotion-detection-fer. Accessed: November 8, 2023. 2020.

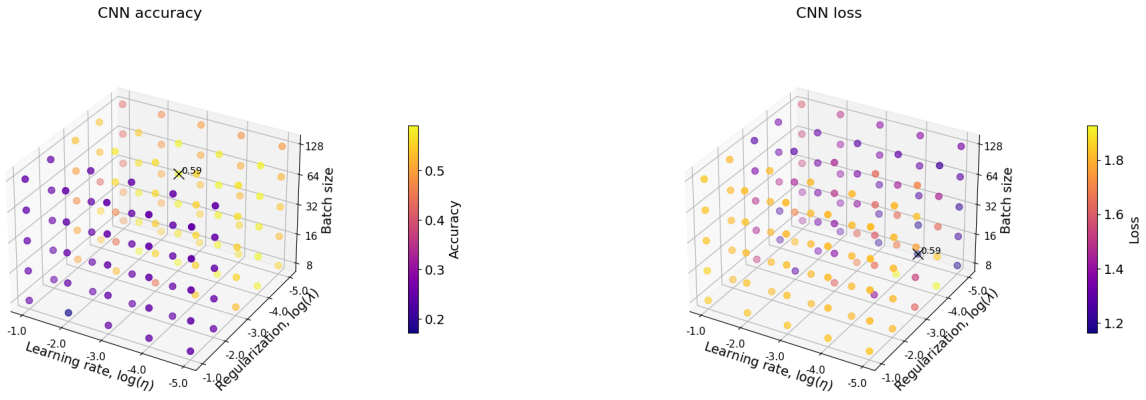
[6] *Machine learning infrastructure (ML Nodes).* University of Oslo, Norway, 2023.

[7] *Tensorflow, Image Data Generator.* https://www.tensorflow.org/api_docs/python/tf/keras/preprocessing/image/ImageDataGenerator. Accessed: November 29, 2023. 2023.

[8] *Facial Emotion Recognition Project using CNN with Source Code.* https://www.projectpro.io/article/facial-emotion-recognition-project-using-cnn-with-source-code/570. Accessed: November 8, 2023. 2023.

[9] *Why Balancing Classes is Over-Hyped.* https://towardsdatascience.com/why-balancing-classes-is-over-hyped-e382a8a410f7. Accessed: December 14, 2023. 2021.

Appendix



(a) **(b)**

Figure 30: The previous 3D grid search scores performed of the learning rate η, L_2 regularization parameter λ and the batch size. The highest accuracy in a) and the lowest loss in b) are achieved with different parameters, which indicate some numerical instabilities.

Table 2: Accuracy score and change of the CNN using XGBoost to take the output of the first fully connected layer as input with different max depths. For comparison, the ordinary CNN model achieved an accuracy of 57.59%.

Max depth	Accuracy [%]	Relative accuracy gain [%]
4	56.85	-1.92
6	57.43	-0.94
8	57.86	-0.19
10	57.70	-0.46
12	56.98	-1.71

Table 3: Accuracy score and change of the CNN using XGBoost to take the output of the first fully connected layer as input with different learning rates, η. For comparison, the ordinary CNN model achieved an accuracy of 56.59%.

η	**Accuracy [%]**	**Relative accuracy gain [%]**
0.05	55.82	-1.35
0.10	56.26	-0.59
0.15	56.55	-0.07
0.20	55.98	-1.08
0.25	56.26	-0.59
0.3	56.35	-0.42

Table 4: Accuracy score and change of the CNN using XGBoost to take the output of the first fully connected layer as input with different boosting rounds. For comparison, the ordinary CNN model achieved an accuracy of 56.73%.

Boosting rounds	**Accuracy [%]**	**Relative accuracy gain [%]**
10	55.48	-2.21
20	56.02	-1.25
30	56.30	-0.76
40	56.28	-0.79
50	56.60	-0.22

Table 5: Accuracy score and change of the CNN using XGBoost to take the output of the first fully connected layer as input with different values of regularization, λ. For comparison, the ordinary CNN model achieved an accuracy of 57.52%.

λ	**Accuracy [%]**	**Relative accuracy gain [%]**
0.0001	56.48	-1.82
0.001	56.44	-1.89
0.01	56.53	-1.72
0.1	56.65	-1.53
1.0	56.87	-1.14

Table 6: Accuracy score and change of the CNN using XGBoost to take the output of the first fully connected layer as input with different values of pruning, γ. For comparison, the ordinary CNN model achieved an accuracy of 57.09%.

γ	**Accuracy [%]**	**Relative accuracy gain [%]**
0.0	57.02	-0.12
0.1	57.45	0.63
0.2	57.36	0.46
0.3	57.38	0.51
0.4	57.43	0.59
0.5	56.87	-0.39