Facial Keypoint Detection Using Convolutional Neural Networks

Zeid Farhat
MA1 Computer Engineering
Université Libre de Bruxelles

***Abstract*—Facial keypoint detection plays a fundamental role in facial analysis applications such as recognition, expression detection, and pose estimation. In this project, we investigate the use of convolutional neural networks (CNNs) to predict 15 facial landmarks (30 coordinates) from grayscale images. We trained our model on a labeled dataset of centered portrait faces, using data normalization and augmentation techniques to improve generalization. The architecture consists of several convolutional blocks with batch normalization and LeakyReLU activations, followed by dense layers for regression. Evaluation was performed using Mean Squared Error (MSE) between predicted and ground-truth coordinates. Testing on external face images dataset showed moderate results when cropped appropriately. This project confirms CNNs' effectiveness for facial keypoint regression when combined with proper preprocessing.**

***Index Terms*—Facial Keypoint Detection, Deep Learning, CNN, Image Processing, Computer Vision**

I. Introduction

Facial keypoint detection is the task of localizing specific characteristic points on human faces, such as the eyes, nose etc... These landmarks are critical for a wide range of applications including facial and emotion recognition, and augmented reality. Accurate detection of facial keypoints is particularly important in fields like biometrics, healthcare diagnostics, and human-computer interaction.

In recent years, deep learning—particularly convolutional neural networks (CNNs)—has emerged as a powerful tool for visual recognition tasks, surpassing traditional methods in both accuracy and robustness.

In this project, We design and train a deep CNN model to predict 15 key facial landmarks (30 (x, y) coordinates) from grayscale facial images. The training data consists of images centered around faces with annotated keypoints. We explore preprocessing steps including normalization, missing data handling, and face cropping using ground-truth landmarks to improve model performance. The trained model is then evaluated on both the original dataset and external test images, with results analyzed both numerically and visually.

This paper details the data preparation pipeline, network architecture, training procedure, and evaluation results. It also discusses the challenges of generalization to unconstrained face images and how preprocessing can significantly impact keypoint detection performance.

II. Dataset and Preprocessing

The primary dataset used for training came from the Kaggle Facial Keypoints Detection competition, which provided us with three CSV files: `training.csv`, `test.csv`, and `id_lookup_table.csv`. The `training.csv` file included both image pixel data (grayscale 96×96) and 30 target values corresponding to the (x, y) coordinates of 15 key facial landmarks. Some entries in this dataset contained missing (null) keypoint values, often due to occlusions or annotation errors. To address this, we applied forward-fill imputation (`fillna(method='ffill')`) to replace missing values using the last non null value before them. Although K-nearest neighbors imputation could have been used, we prioritized simplicity for this implementation.

The `test.csv` file included only pixel values, with no target labels, while the `id_lookup_table.csv` specified which keypoints were required for submission in the original competition. However, to properly evaluate our model, we used an additional labeled dataset containing true keypoint annotations for the test images. This allowed us to compute error metrics such as Mean Squared Error (MSE), visualize predictions, and compare them to ground-truth coordinates.

Since the CNN was trained on tightly cropped, centered facial images of size 96×96, we ensured consistency when evaluating on external images by preprocessing them accordingly. This involved converting color images to grayscale, and cropping the face region using the ground-truth keypoints from XML annotation files. We computed a bounding box around the keypoints and applied a margin before resizing the cropped face to match the expected input dimensions. This approach significantly improved prediction quality by ensuring alignment with the training data distribution.

After handling missing values, we separated the dataset into features and labels. The image pixel values served as features, while the remaining columns represented the target keypoints to predict. Because the pixel data was initially stored as strings of space-separated values, we converted each string into a NumPy array and reshaped it to form 96×96 grayscale images. These were later flattened and normalized to fall within the $[0, 1]$ range before being fed into the model.

An example of a training image is shown in Figure 1.

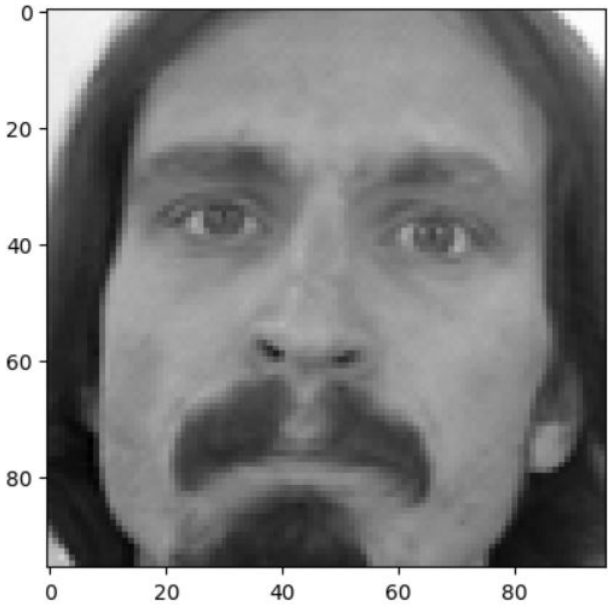

Fig. 1: First picture of the training set [1]

III. Model Architecture and Training

The goal of our model was to predict the (x, y) coordinates of 15 facial keypoints (30 values in total) from grayscale facial images of size 96×96 [1]. We implemented a deep convolutional neural network (CNN) using the Keras API with TensorFlow as the backend. The model was built using the `Sequential` class, allowing a clear, layer-by-layer construction.

A. Neural Networks and CNNs

A neural network is a computational model inspired by the human brain, composed of layers of interconnected units called *neurons*. Each neuron receives one or more input values, applies a learned *weight* to each input, sums them, and passes the result through a non-linear *activation function*. This activation determines whether the neuron "fires," introducing non-linearity into the model and enabling it to learn complex patterns.

The network is organized in layers: an input layer, one or more hidden layers, and an output layer. During training, the network adjusts its weights through a process called *back-propagation*, which computes gradients of a loss function with respect to each weight using the chain rule. These gradients are then used to update the weights via an optimization algorithm, such as gradient descent. Over multiple iterations (epochs), the model learns to map inputs (images) to the correct outputs (keypoint coordinates) by minimizing the error between predicted and true values.

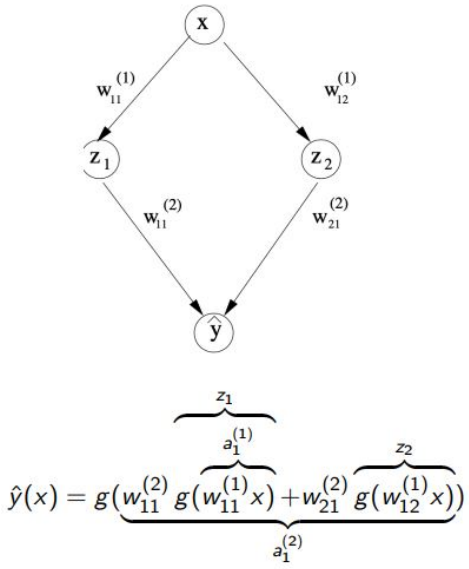

Fig. 2: Illustration of a simple neural network with one hidden layer. The input x is passed through two neurons with weights $w_{11}^{(1)}$ and $w_{12}^{(1)}$. Each neuron applies an activation function $g(\cdot)$, producing z_1 and z_2. These are combined with weights $w_{11}^{(2)}$ and $w_{21}^{(2)}$ in the output layer. The final prediction $\hat{y}(x)$ is again passed through an activation function. This shows how a neural network learns by adjusting weights and applying non-linear activations to model complex functions. [7]

A *Convolutional Neural Network* (CNN) is a specialized type of neural network designed for image data. Instead of learning from raw pixel values with fully connected layers, CNNs use *convolutional layers* that apply small learnable filters across the image to detect local patterns such as edges, corners, or textures. These filters slide over the input to produce *feature maps*, which preserve spatial structure and significantly reduce the number of parameters compared to traditional dense layers. By stacking multiple convolutional and pooling layers, CNNs can extract increasingly abstract representations of the image, making them highly effective for visual tasks like facial keypoint detection.

B. Architecture Design

The model began with a series of convolutional blocks. Each block consisted of two convolutional layers with 3×3 kernels and `padding='same'` to preserve spatial dimensions. Bias terms were omitted in the convolutional layers since batch normalization layers followed, and they included their own trainable parameters and offsets.

Each convolutional layer was followed by a LeakyReLU activation function, a variant of the standard ReLU (Rectified Linear Unit). While ReLU set all negative outputs to zero, LeakyReLU allowed a small, non-zero gradient for negative inputs, helping to avoid inactive neurons during training. [1]

We applied batch normalization after each activation to standardize intermediate outputs, which improved training stability and accelerated convergence.

After every two convolutional blocks, we inserted a max pooling layer with a pool size of $(2, 2)$ to downsample the feature maps. Max pooling is a downsampling operation that slides a fixed window (typically 2×2) over the input and selects the maximum value from each region. This reduced the spatial resolution of the feature maps while retaining the most salient features. It also lowered computational cost and helped the model focus on the most informative regions. [3]

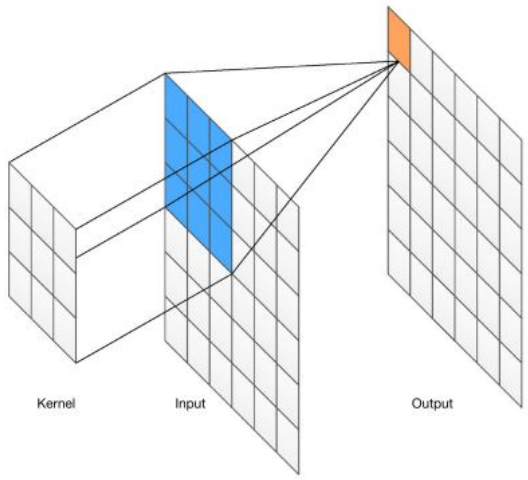

Fig. 3: An example of how the output is computed through a convolutional layer. The kernel is the filter. [3]

The model started with 32 filters and progressively increased the depth through layers of 64, 96, 128, 256, and finally 512 filters. This increasing depth allowed the network to learn hierarchical feature representations, from low-level edges to high-level face-specific structures.

Once the final convolutional block was completed, the feature maps were flattened and passed to a fully connected dense layer with 512 units and ReLU activation. The dense layer's weights were optimized using gradient descent, an iterative algorithm used to minimize the model's loss function. At each step, the model computes the gradient of the loss with respect to the weights and adjusts them in the opposite direction of the gradient. This process gradually reduces the prediction error by finding weight values that improve the model's performance. In our case, we used the Adam optimizer, which is an extension of gradient descent that adapts the learning rate for each parameter, leading to faster and more stable convergence. To prevent overfitting, we applied a dropout layer with a rate of 0.1 after the dense layer. This means that at each training iteration, 10% of the neurons are randomly deactivated, helping the model generalize better by not relying too heavily on specific neurons. Overfitting occurs when a model learns to memorize the training data rather than generalizing to new, unseen inputs. This is especially common in deep networks with many parameters and relatively small training datasets [1]. This prevented the network from becoming overly reliant on specific neurons and encouraged it to learn more robust, distributed representations. At test time, all neurons were used but their outputs were scaled accordingly to maintain consistency with training.

The final layer was a dense layer with 30 units (corresponding to the 30 coordinate values), with no activation function, as this was a regression task.

C. Output and Label Scaling

Because the original keypoint values ranged from 0 to 96 (image width/height), we normalized the labels by dividing all coordinates by 96 prior to training. This scaled the targets to the $[0, 1]$ range, which improved convergence and model stability. After prediction, the model's outputs were rescaled back by multiplying by 96 to retrieve the original coordinate scale. This normalization step was important in deep regression tasks involving bounded outputs.

D. Data Augmentation

To increase robustness and help the model generalize to unseen facial images, we applied data augmentation using the `ImageDataGenerator` class from Keras. The augmentations included random rotations (±15 degrees), horizontal and vertical shifts (up to 10%), zooming (±10%), shearing, brightness variation between 80% and 120%, and horizontal flipping. These transformations simulated common variations in real-world images, such as lighting conditions, facial orientations, and minor misalignments, allowing the model to become more invariant to such distortions.

Augmentation was applied on-the-fly during training, ensuring that the model saw a slightly different version of each image at every epoch. This greatly reduced the risk of overfitting and improved the model's ability to generalize.

E. Training Procedure

The final model was compiled using the Adam optimizer with default parameters and trained using the MSE loss function, which was suitable for this regression task. It quantifies the average of the squared differences between the predicted and actual keypoint positions. The MSE is computed as shown below [3]

$$e = \frac{1}{N} \sum_{i=1}^{N} (y_i - \tilde{y}_i)^2 \tag{1}$$

Equation (1), where N is the total number of keypoint coordinates, y_i is the true value, and $\tilde{y}_i$ is the predicted value.

The model also tracked the mean absolute error (MAE) during training as an additional performance metric. The MAE is another commonly used metric in regression tasks. It calculates the average of the absolute differences between predicted and actual values. It is less sensitive to outliers compared to MSE and is defined as [3]:

$$\text{MAE} = \frac{1}{N} \sum_{i=1}^{N} |y_i - \tilde{y}_i| \tag{2}$$

MAE is more easily interpretable than MSE but it wasn't used for training To evaluate generalization during training,

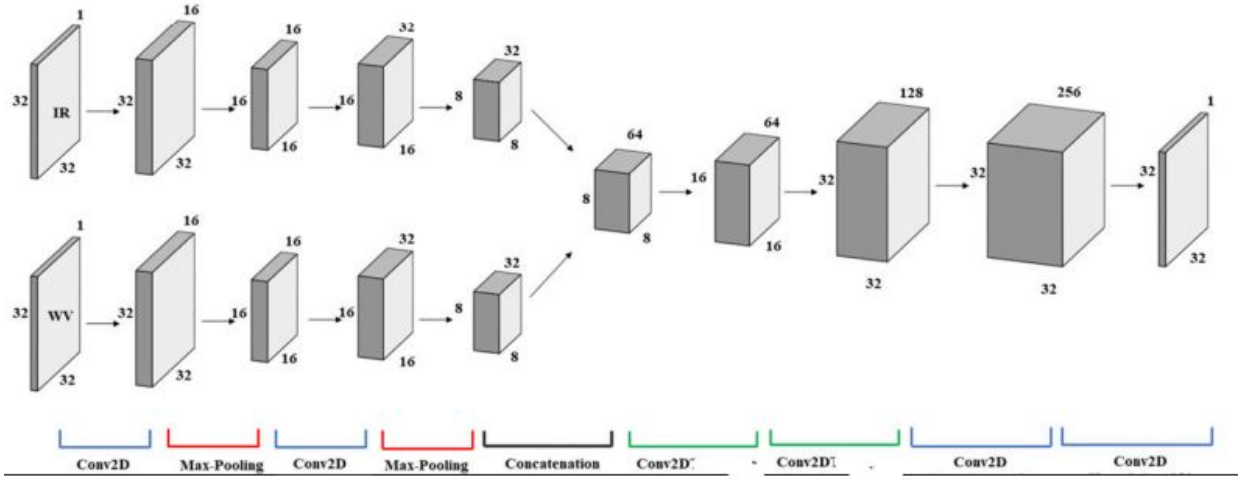

Fig. 4: Visual representation of the chain of filters and operations in a CNN [4].

we split the dataset into 80% training and 20% validation sets using `train_test_split`. We then trained the model using the `fit()` function, passing in the augmented training data generated by `ImageDataGenerator`, and manually specifying the validation data.

Training was performed over 150 epochs with a batch size of 256. Throughout the training process, we monitored the validation loss to ensure that the model was not overfitting. [6] If necessary, early stopping or learning rate adjustments could be explored in future work to further improve performance.

IV. Evaluation and Results

Since this was a regression task, model performance was measured using the MSE between predicted and true keypoint coordinates. The model outputs were first rescaled to the original pixel range by multiplying by 96. Although training and validation errors remained consistently low, some individual keypoint predictions on test images were slightly inaccurate. This was expected, as the model was trained on tightly cropped, centered facial images and may struggle when applied to more varied inputs with different lighting or alignment.

The close match between training and validation loss throughout training suggested that the model generalized well. For instance, in the final epoch, the training loss and MAE were 0.0016 and 0.0270, while the validation loss and MAE were 0.0011 and 0.0214, respectively. These nearly identical values confirmed that the model did not overfit and learned a stable mapping from image to keypoints. These values have to be multiplied by 96 as all pixel values were normalized before training. This means the error for validation was around 2 pixels on average.

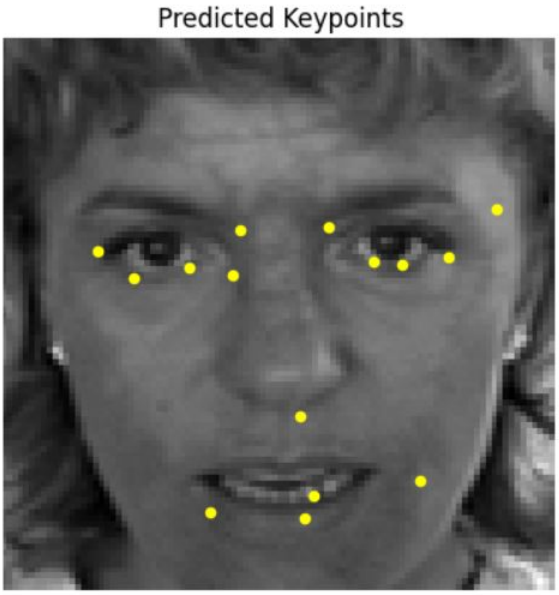

Fig. 5: Example of predicted facial keypoints (yellow) on a test image from the competition dataset [1].

To further assess generalization, we tested the trained model on an external dataset consisting of facial images not seen during training. These images varied in background, face size, and lighting conditions. Additionally, faces were not always centered and were sometimes rotated along the vertical axis. Since the model was trained on aligned grayscale images of size 96×96, we first preprocessed each external image by cropping around the face using provided keypoint annotations from an XML file, applying a fixed margin, and resizing to match the training input format.

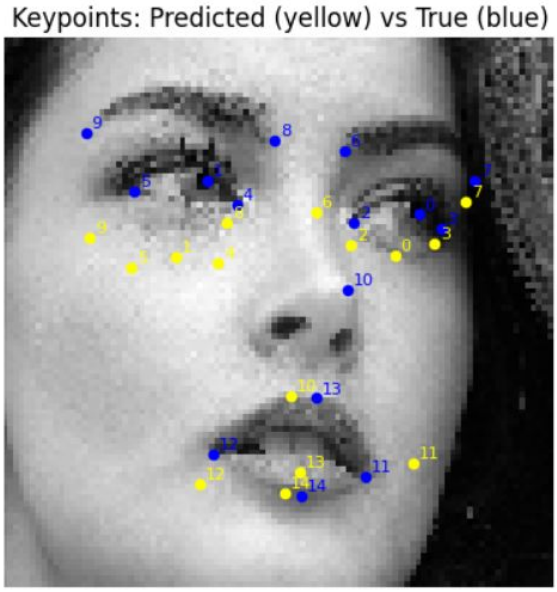

Fig. 6: Example of predicted facial keypoints (yellow) vs true keypoints(blue) on a test image from the external dataset. [2]

We then computed the MSE across the external test images by comparing the model's predictions to the true coordinates, after aligning both to the cropped 96x96 frame. The resulting MSE was approximately 216 squared pixels for the first image of the set, corresponding to an average error of about 15 pixels per coordinate. This was considerably higher than the validation error, highlighting the model's difficulty with images that differ from the training set in alignment, scale, or background. This performance gap could have been reduced by including more diverse examples during training, such as rotated or off-centered faces. Training the model for more epochs should also be considered.

It should also be highlighted that for every image in the new dataset, the keypoint coordinates had a different ordering. This forced us to hard code the MSE for every image. Despite the increased error, visual inspection showed that the predicted keypoints still roughly followed the correct facial structure. As shown in Figure 6, most keypoints were positioned near their correct locations, indicating that the model was still able to extract meaningful features from unfamiliar inputs when appropriately preprocessed.

V. Conclusion

In this project, we implemented a convolutional neural network to perform facial keypoint detection on grayscale images. The model was trained on the Kaggle dataset using a deep architecture and regularized with data augmentation techniques to improve generalization. Evaluation on the validation set showed strong performance, with low MSE and consistent results across epochs, indicating stable and effective learning.

To further assess generalization, we tested the model on an external dataset with face images under less controlled conditions. While the validation performance was strong, the average error increased significantly on these new images, primarily due to differences in image alignment, scale, and background. Cropping the face based on ground-truth keypoints before inference helped reduce this mismatch, but the results highlighted the limitations of training solely on clean, centered data.

Overall, the model demonstrated the ability to learn meaningful spatial features and generalize to new faces when sufficient preprocessing was applied. Future improvements could include training on a more diverse dataset, fine-tuning on a subset of external examples, or using face detection techniques to better automate the cropping step. This would improve robustness and bring the model closer to real-world applicability.

References

[1] Karan Jakhar, *Facial Keypoints Detection*, Kaggle, 2019. Available at: https://www.kaggle.com/code/karanjakhar/facial-keypoint-detection

[2] Karan Jakhar, *Facial Keypoints Detection - Photos & Labels*, Kaggle, 2023. Available at: https://www.kaggle.com/datasets/trainingdatapro/keypoints

[3] S. Wu, J. Xu, S. Zhu, and H. Guo, "A Deep Residual Convolutional Neural Network for Facial Keypoint Detection with Missing Labels," *Signal Processing*, vol. 144, pp. 384–391, 2018. Available: https://doi.org/10.1016/j.sigpro.2017.11.003

[4] M. Sadeghi, A. A. Asanjan, M. Faridzad, P. Nguyen, K. Hsu, S. Sorooshian, and D. Braithwaite, "PERSIANN-CNN: Precipitation Estimation from Remotely Sensed Information Using Artificial Neural Networks–Convolutional Neural Networks," *Journal of Hydrometeorology*, vol. 20, no. 12, pp. 2273–2289, 2019. Available: https://doi.org/10.1175/jhm-d-19-0110.1

[5] Y. Omer, R. Sapir, Y. Hatuka, and G. Yovel, *What Is a Face? Critical Features for Face Detection*, Perception, vol. 48, no. 5, pp. 437–446, 2019. doi: https://doi.org/10.1177/0301006619838734.1177/0301006619838734.

[6] G. Bontempi, *INFO-F422: Statistical Foundations of Machine Learning*. Version unknown. Brussels: ULB, 2025. [Online]. Available: https://uv.ulb.ac.be/pluginfile.php/4093028/mod_resource/content/11/nonlin.pdf. Accessed: May 4, 2025.

[7] G. Bontempi, *INFO-F422: Statistical Foundations of Machine Learning*. Version unknown. Brussels: ULB, 2025. [Online]. Available: https://uv.ulb.ac.be/pluginfile.php/4093030/mod_resource/content/13/algos.pdf. Accessed: May 4, 2025.