SOC FINAL PROJECT REPORT

Name – Pratham Bagdi Roll No – 24b0700

In this project , I built facial emotion detection model through Convolutional Neural Networks. My approach involvement various steps like Data Preprocessing , Data Augmentation , Splitting the Data , Trying different CNN Architectures and finally fine-tuning the model. Brief description of each step is given below-

1) **Loading the Dataset –** I used the FER 2013 dataset , which consisted of 48 x 48 greyscale images and over 28k different entries. The code for loading was already provided to us by our mentor.
2) **Data Preprocessing –** In data preprocessing I normalized the pixel values by calling an helper function , also I performed Data Augmentation on small subset of training sample after performing train-test-split.
3) **Data Splitting –** For splitting my data into train , test and validation datasets I used train-test-split function of scikit learn for generating the random indices of entries to be stored in each type of dataset. Further , I sampled some random indices from my training sample and performed Data Augmentation for images of those indices and merged the images corresponding to indices of original training sample with augmented images to generate my final training sample . Similarly , test and validation datasets were also generated but without Data Augmentation. Lastly, I stored my training , test and validation datasets so that I can directly load them whenever needed further in my model.
4) **Model Building –** This was the most challenging step of entire project , I tried many models ranging from Simple CNNs to Hyperparameter tuned and pretrained models , to get the best results.

   Firstly I started with a generalized CNN architecture with 4 layers involving features like **Dropouts (Reduces the number of active nodes/ active filters in hidden/convolutional layers , depending on provided ratio) and Batch Normalization (Normalizing Inputs for each hidden/convolutional layer)** which helped to reduce overfiitting of the model. The key details behind these model and also my further models were **– 1) The filters in each convolutional layer either increases progressively or remains constant** , this is because the initial layers of a CNNs identify features which are superficial and common to many images , hence they requires less filters as compared to future layers which capture more complex and abstract features and hence require more filters. **2) The dropout rate keeps on increasing progressively as we go deep in the network ,** this is because initial layers capture features that are superficial and common to many images hence more active filters are required . On the other hand , since deeper layers capture more complex and image specific features , increasing dropout ratio helps to decrease active filters and further reducing overfitting of model by preventing it to learn from input specific features.

The accuracy of my first model on test dataset was 53.43 %.

In next model, I introduced **hyperparameter tuning using KerasTuner**, which allowed the model to automatically search for the best values for filter sizes, dropout ratios, dense units, and even the learning rate. This not only made the model more flexible but also ensured that it didn't rely on fixed heuristics and instead adapted to the dataset characteristics.

I retained a 3-block CNN structure but each block's parameters—like the number of filters and dropout rate—are now tunable. After the convolutional layers, **GlobalAveragePooling2D** was used instead of Flatten(). This layer compresses the spatial dimensions by taking average values across each feature map. Compared to flattening, it drastically reduces the number of parameters and adds a mild regularization effect by enforcing feature abstraction rather than memorization. This helped prevent overfitting without sacrificing performance.

Pooling after each conv block is still done using **MaxPooling2D**, which plays a vital role in reducing spatial dimensions while keeping the most prominent features. It basically ensures that only the strongest activations (features) are retained, which makes the network more translation-invariant and computationally efficient.

The final classification is done using a dense layer with **softmax** activation, which is standard for multi-class problems. The model uses **sparse categorical crossentropy** as the loss function, and that choice is made because the labels are integers (not one-hot encoded). This saves memory and makes training more efficient while achieving the same result.

Learning rate is also treated as a tunable parameter here. Instead of hardcoding it, I allowed the tuner to choose it from a logarithmic scale, enabling finer control—especially important because learning rate has a strong impact on convergence speed and stability.

The accuracy of this model was 53.01 % on test dataset.

My best model incorporated a blend of both architectural depth and training robustness. Unlike earlier experiments, this one was built with **extensive regularization, advanced activation, and aggressive data augmentation**, all of which combined to push performance to its highest.

The model starts with five convolutional blocks, each padded to preserve spatial dimensions. Instead of ReLU, I used **LeakyReLU** activations with a small alpha, which helps prevent the issue of dying neurons—a common problem in deep CNNs when too many neurons output zero due to strong negative activations. It maintains a small gradient even when the unit isn't active, improving gradient flow especially in deeper layers.

A **significant bump in filter sizes** starts right from the beginning with 128 filters, progressively rising to 512 in the deepest layer. The intention was to allow the network to learn high-capacity representations, which complements the model's deeper structure. Alongside this, **Dropout rates also vary**, placed strategically after each block to cut down on overfitting while still allowing the model to learn rich representations.

Data augmentation was another crucial addition. With transformations like rotation, shifts, zoom, shear, and horizontal flips, the training set became much more diverse. This gave the model resilience against minor visual distortions and improved its generalization capability significantly.

For final feature abstraction, **GlobalAveragePooling2D** was retained. Again, it outperformed flattening by reducing overfitting, and worked well with high-dimensional filters in the later layers. A dense layer with 512 units follows, equipped with LeakyReLU and further regularized with both BatchNorm and Dropout (set to 0.5 here, the highest in the model) to stabilize and smooth out the final decision space.

Label smoothing was introduced through **CategoricalCrossentropy with smoothing=0.1**, which helped the model handle slightly noisy labels better and also made the softmax predictions less confident—resulting in improved calibration. This was paired with **one-hot encoded targets** because this time the loss function expected class probabilities instead of class indices.

The optimizer used was Adam with an initial learning rate of 1e-3, but I added a **ReduceLROnPlateau** callback to automatically drop the learning rate when validation accuracy plateaus, giving the model more time to fine-tune weights in later epochs. Together with **EarlyStopping**, the training was made both efficient and adaptive.

Overall, this was the most performant version of my CNN pipeline so far and I saved it for final deployment. The accuracy of this model after training for several epochs was 62 %.

Further, I experimented with a few pretrained architectures like **ResNet** and **EfficientNet**, aiming to leverage transfer learning for potentially better performance. However, they came with a set of complications—most notably around **image resizing requirements**, **dimensional mismatches**, and **heavy RAM consumption** during training. Even after resolving the shape-related and preprocessing issues (like converting grayscale 48×48 inputs to RGB 224×224 format), the models didn't perform well in practice. In fact, after all the adjustments, their validation accuracy dropped and the training became unstable or too resource-intensive to be feasible. So despite the theoretical promise of pretrained models, I chose not to include them in my final pipeline due to their **underwhelming real-world performance.**