**Deep Learning for Facial Emotion Recognition: A Comparative Analysis of CNN Architectures**

**Google Drive link:** <https://drive.google.com/drive/folders/13sQ5ci_s75Mb-WwCjqq7NMsCPFRlebKD?usp=sharing>

**Abstract**

The face expression recognition is one of the most complicated problems in computer vision for making strong feature extraction and classification techniques. This paper tells about work done basically on the development of a convolutional neural network (CNN) architecture to classify temporal gray images under two categories: "happy" and "sad." The first model was a very simple CNN that was made with just two convolution layers and Sigmoid activation functions, so unsurprisingly, it did not perform well at all, having an associated loss of 0.6931 and an accuracy of barely 50 percent. It was later changed to ReLU, which hugely improved the speed of convergence and learning of the model, bringing down the loss value to 0.3452 and the accuracy value to 79 percent. At the third iteration, the model was further tuned by increasing convolutional filters and applying Adam optimizer, which resulted in a dramatic jump in performance ending with a loss of 0.0001, and the accuracy was 84.2 percent. Also, the infrastructure changes from AWS to HDFS and over different TensorFlow incompatibilities by swapping to PyTorch significantly improved within the system and reflected upon its impact. All these findings combine evidences of the fact that optimized CNN architectures work and also give importance to activation functions, optimizers, and infrastructure choices in really deep learning applications.

**1. Introduction**

It involves recognizing facial expressions in artificial intelligence. Examples of applications include human-computer interaction and emotional analysis in psychology. Deep learning through convolutional neural networks has changed the whole practice regarding its applications in automatic feature extractions and classifications from image datasets. Unfortunately, optimum channel architectures have continued to challenge maximized accuracy in computational efficiency.

This study aims to develop a CNN-based facial expression recognition model to analyze grayscale images for the "happy" and "sad" categories. In the first phase, a simple CNN model with two convolutional layers was tested, and Sigmoid was used as the activation function, and Stochastic Gradient Descent (SGD) was the optimizer. Nevertheless, the model cannot perform because of the vanishing gradient problem relating to Sigmoid. Therefore, to prove this point, ReLU was introduced as an activation function during the second iteration. Appallingly, time for training improved considerably and accuracy increased. In the last test, the convolutional filters were increased and the Adam optimizer was used to achieve quicker convergence and better performance.

There were also some infrastructure challenges during the development of the application involved here: it was initially designed to use AWS for cloud storage but was scrapped because the connection did not last long, and finally, developers turned to HDFS for local storage. It also started with TensorFlow for building the application model but was re-directed to PyTorch due to hardware incompatibility because PyTorch has greater flexibility and computation efficiency; these changes at architectural, algorithmic, and infrastructural levels all tuned for improved performance of the model.

**2. State of the Art**

Recognizing people's expressive faces or facial expression recognition (FER) is one of the applications or one of the tasks of computer vision. It is increasingly applied in relation to human-computer interaction, psychology, and security. Existing methods rely on traditional feature engineering methods which use Local Binary Patterns (LBP) or Histogram of Oriented Gradients (HOG), and such approaches could not deal with lighting, pose, and occlusion variations.

Deep learning, and in particular, the use of Convolutional Neural Networks, has taken over for FER since it allows automatic extraction of the spatial hierarchies of features. CNNs have outperformed traditional methods in benchmark datasets such as FER-2013 and CK+. Some of the popular architectures for emotion classification are VGG, ResNet, and AlexNet, but these architectures have been found to perform well with high accuracy. A downside of CNN usage is that it needs a lot of computing power; for example, they often rely on GPUs for efficient training. In this study, however, training was done on CPUs owing to hardware limitations within a virtual machine (VM).

In order to present a solution for efficiently managing large-scale image datasets, the Hadoop framework and distributed file system (HDFS) have also been integrated into the workflow. Hadoop is suited to extend its scalable storage and processing of image data across many nodes in a cluster, which is important in the case of handling very large amounts of training images. With the integration of PyTorch - a flexible deep learning framework - and distributed storage provided by Hadoop, their combination allows quicker data loading and model training. For example, PyTorch has an excellent dynamic computation graph which makes it easy-to-use.

Challenges Making Faces Expressive Yet Realistic by Highly Advanced Recognition Technologies in FER Still Generate Great Lots of Experimentation Efforts. It also involves model generalization to outside real-world conditions and reduction of the computational cost. This study would like to explore them by formulating and testing a CNN architecture trained on facial expression datasets contained in HDFS.

**3. Literature Review**

The research reveals how deep learning, especially Between Convolution Neural Networks (CNNs), occupies a pivotal place in image classification through different keys; domains include healthcare (skin cancer diagnosis, face mask detection), human emotion analysis, and agriculture (plant recognition).

Key among the highlighted salient points in the papers include:

CNN architectures in varied configurations proved to be their hub to achieve such high performance in the image classification tasks. Different architectures like ResNet, Inception (GoogLeNet), and EfficientNet have superiority in various ways like accuracy, efficiency, and enable their training in very deep networks. It is nowadays also recognized as two major architectural innovations compared to the ones that were previously used by ResNet introducing residual connections in order and Inception using parallel processing by the module.

PyTorch is most often used as deep learning framework implementation and deployment. The comparison of PyTorch and TensorFlow could consider trade-offs between user friendliness and ease of integration, training speed, and accuracy, all eventually leading to a situation where the framework choice varies with the particular project context.

Data is, indeed, essential for the success of the deep learning model. The authors argue for large and diverse datasets, further adding that augmenting data is necessary for model robustness and generalization capability-which is imperative when data is scarce.

Transfer learning using pre-trained models such as Inception v310 is viewed particularly as an effective means of transferring existing knowledge into improved performance, particularly when the model is subjected to training with a domain-specific dataset.

This study basically concentrates on designing and developing a highly efficient and accurate image classifier, which may be useful in real-world scenarios such as automated diagnosis in medicine, public health monitoring, and even agricultural automation.

The papers summarize the capability of deep learning using CNNs in solving complicated image classification problems spanning several disciplines while also including the technologies of considering frameworks and data management along with design considerations.

**4. Methodology**

The Original idea for thisapplication was cloud-enabled with AWS to give maximum scalability with respect to dataset storage and processing. But then connectivity errors rudely interrupted the project due to difficulty in establishing a connetion with the AWS servers even after giving all the permissions and creating the access keys. After a few futile attempts to reactivate the connection, I decided that an even more favorable option would be storing data locally in a Hadoop Distributed File System, giving reasonable access to the data set while being efficiently distributable and manageable.

File data upload on HDFS was made with no issue and and for the data access and manipulation my original plan was to use PyArrow, but it generated issues like HDFS connection error and after running through a few permutations of the possible solutions for this issue, the fsspec (filesystem specification) interface had emerged as the right solution: A more flexible and highly capable filesystem interface that allows a much easier way to access data from HDFS.

TensorFlow was chosen for the construction and deployment of the model as it was the most commonly used and powerful deep learning framework capable of addressing large-scale machine learning tasks. Unfortunately, just when it was being put to use in this project, TensorFlow had an untimely exit from the computing environment as incompatibility issues showed up due to the missing AVX (Advanced Vector Extensions) support as it is not supported in my Virtual Machine. PyTorch was therefore adopted as a dynamic computational graph-based deep learning framework, easy to learn, and highly compatible with numerous hardware configurations. Such operational convenience proved to be a vivid separating benefit in development ease and high performance in training and deploying the deep learning model [4].

A further major issue happened in the form of the kernel crashing with the message, "Kernel has died and will restart." Further investigation revealed this to be a resource-related problem. Therefore, the decision arose to increase the VM memory from 4GB to 12GB and increase the size of the VM hard disk from 120GB to 200GB after determining that a larger HDFS storage capacity indeed would be required. These upgrades greatly aided system stability, allowing the project to advance with no more interruptions.

### 

### **4.1 Data Preprocessing and Loading**

Before feeding the images into the CNN models, preprocessing steps were required. These included:

* **Resizing images** to a fixed dimension to ensure uniformity.
* **Normalization** by scaling pixel values to the range [0,1], which helps in stable gradient updates.
* **Data Augmentation** techniques such as flipping and rotation (applied to prevent overfitting in the model 3).
* **Splitting into training and validation sets** to assess model generalization.

### 

### 

### **4.2 Model Architectures**

To create a better image classification system, I have carried out implementation of three different Convolution Neural Network (CNN) architectures. These all models classify the facial expressions into two categories that are happy and sad. These two classes formed the basis for interpreting the predictions of the model in a human-readable way.

In fact, using PyTorch, there is therefore the normal procedure for defining network architectures through subclassing the module of torch.nn because this is the superclass of all PyTorch models. One of the many advantages that PyTorch has over other competing software libraries is that it allows dynamic computation graphs in setting up and debugging models. Automating the differentiation by torch. Autograd facilitates speedy backpropagation and elimination-developer-friendly. Furthermore, metals are optimized for GPU acceleration to provide more utilizing speed in training for larger datasets.

Unfortunately I couldn't use the GPU acceleration because my virtual machine (VM) environment has certain limitations. Instead, I had to depend on CPU training, which, although possible, slowed down the training process. Nevertheless, the simplicity of use and the strong debugging capabilities of PyTorch have made innovations in the architecture of different CNNs very much easier to come up with.

**4.2.1 First model**

I decided to prepare a simple model as my first test. Just to see how it will deal with the data. I will insert 2 convolutional layers and one max pooling layer

In the first convolutional layer performs feature extraction from the input images. The layer parameters are:

* **Input channels (1)**: Since the images are grayscale, they have 1 channel (rather than 3 channels for RGB images).
* **Output channels (6)**: This indicates that the network will use 6 filters (kernels) to extract 6 different feature maps.
* **Kernel size (5)**: The kernel is a 5x5 matrix, and it slides over the image to perform the convolution operation [7].

This layer extracts fundamental features, like edges and textures, from the input image. The use of multiple filters allows the model to capture different aspects of the image.

Max pooling is a downsampling operation that reduces the spatial dimensions of the feature maps, effectively reducing the computational load and allowing the model to focus on the most important features [2].

In this case, the pooling operation uses Stride; in this case I define it as 2, so it will be a 2x2 window, meaning it downsamples the input by a factor of 2, keeping the maximum value in each 2x2 region. This helps preserve the most unnecessary features while reducing the image size and computational complexity.

The second convolutional layer operates on the output of the first convolutional layer. The parameters are:

* **Input channels (6)**: Since conv1 produces 6 feature maps, these become the input to conv2.
* **Output channels (16)**: This layer uses 16 filters to produce 16 feature maps.
* **Kernel size (5)**: Again, a 5x5 filter is used to extract features from the input.

This second convolutional layer helps the model learn more complex patterns by combining the features learned in the first layer, resulting in higher-level abstractions of the image.

The next step is to flatten the feature maps and pass them through a series of fully connected layers (FC layers). These layers help map extracted features to the final classification output.

Before passing the data to an FC layer, the 2D feature maps produced by convolutional layers must be converted into a 1D vector.

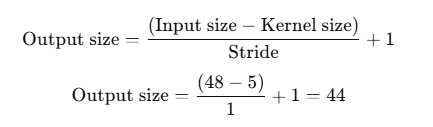
self.flattened\_features = 16 \* 9 \* 9  *# 16 channels × 9×9 spatial dims*

Why 9\*9? It comes from the result of Conv1(first layer) + maxPool + Conv2(second layer) + maxPool.

In the Conv1, we have:

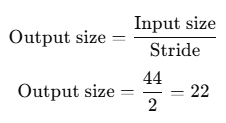
self.conv1 = nn.Conv2d(1, 6, 5) *# 1 input channel, 6 filters, 5x5 kernel*

The input size is equal to the image size, which is 48 X 48



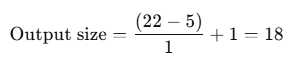
Then, after the Conv1, we have our maxPool, and it will receive the output from conv1: 44. In maxpool we must decide the size of our stride,

self.pool = nn.MaxPool2d(2, 2)



In our second conv (Conv2), we receive the output from maxpool: 22

self.conv2 = nn.Conv2d(6, 16, 5) # 6 input channels, 16 filters, 5x5 kernel



And finally, our last maxpool will receive the conv2 output: 18



By this, we can return to our flatten formula.

*self.flattened\_features = 16 \* 9 \* 9*

Just a reminder, 16 comes from Conv2 Layers output.

#### Fully connected layers is the part where the neural network starts to recognize general patterns related to the emotions. The choice of three fully connected layers in my CNN is based on progressive feature refinement and historical deep learning architectures (like LeNet-5, AlexNet, etc.).

self.fc1 = nn.Linear(self.flattened\_features, 120)

self.fc2 = nn.Linear(120, 84)

self.fc3 = nn.Linear(84, 2)

* **fc1**: The first fully connected layer takes the flattened feature vector and transforms it into 120 units. This layer learns higher-level representations of the features.
* **fc2**: The second fully connected layer reduces the dimensionality further, from 120 units to 84 units.
* **fc3**: The final fully connected layer produces 2 output units, corresponding to the two classes ('happy' and 'sad'). For this reason, the number 2 is the output number in fc3.

Fully connected layers help the model make the final decision by combining the learned features into a final classification score.

When applying the defining method I will use Sigmoid [9] just for testing how it will behave because Sigmoid is not a good fit for this method due to the vanishing gradient issue. The gradient becomes very small, preventing the network from learning effectively. Also, it returns a probability between 0 and 1.

Before we apply all of it to our training loop we need to select our Loss function and our Optimizer.

Regarding our Loss function, we will use CrossEntropyLoss because it is the loss function commonly used for binary classification tasks, in this case: Happy or Sad. For our optimizer, I will test it first with Stochastic Gradient Descent [6] or SGC, this is an optimization algorithm used to update the weights (parameters) of a neural network to minimize the loss function. It is a variation of traditional Gradient Descent, but instead of computing gradients over the entire dataset at once, SGC updates the model parameters after processing each mini-batch or even a single training example.

This optimizer is required because neural networks learn by adjusting weights to reduce the difference between predicted and actual values (i.e., minimizing the loss function). To do this, we calculate the gradient (or slope) of the loss function for each weight and update them in the opposite direction of the gradient to move toward an optimal solution.

As I was expecting, the result wasnt very good. Loss: 0.6931 and accurancy of 50% only.

Just for a better understanding of what happens to an image in the first convolutional layer of my model. To this end, I first set the model to evaluation mode using net\_test1.eval() to ensure proper functioning during inference of layers such as batch normalization and dropout (if any). Since no gradients are required in these operations, I use a context manager with torch.no\_grad() to save memory and time.

Then, I select one image from the dataset and reshape it using unsqueeze(0) to create a batch dimension—the shape is now (1, channels, height, width) to suit the model input. For this reason, the actual images are grayscale; thus, the input starts as (1, 48, 48), and upon unsqueezing, changes to (1, 1, 48, 48), 1 denoting the batch size and the number of channels.

I then forward the image through conv1 (the first convolutional layer of my model) that applies six filters to generate six feature maps of size (6, 48, 48). For easier visualization, I squeezed the batch dimension using squeeze(0), thus changing the shape of the tensor to (6, 48, 48). Since matplotlib requires plotting with NumPy arrays, I translate the feature map PyTorch tensor into a NumPy array.

Finally, I plot the six feature maps in place using plt.subplots(1, 6). Each subplot shows the output of each specific filter applied to the original image. This visualization allows me to understand how the convolutional layer extracts different features from the input, emphasizing edges, textures, and patterns, which will be employed by the model in subsequent layers.



**4.2.2 Second model**

Previously, I made use of the Sigmoid activation function for the test. In this test, however, I have decided to use the activation function ReLU (Rectified Linear Unit) for several reasons. The main reason for changing to ReLU is that it circumvented to some degree the vanishing gradient problem, which is very often realized with the Sigmoid function [3] [5]. The Sigmoid function drive the gradients to very small values backward in backpropagation, especially in very deep networks, thus necessitating the model to update the weights with less efficacy. This results in slow convergence and long training times.

ReLU does this by allowing all positive values to pass as-is, while negative values are set to zero. Being simple but efficient, this non-linearity allows for the better maintenance of a larger gradient during training and hence faster convergence and improved learning. This suits the deep networks more where vanishing gradients come into play heavily. The ReLU activation function applied in the convolutional layers in both these models is essential so that the network can effectively learn and transmit important features down the layers.

Switching to ReLU has decreased the loss of the model to 0.3452 and increased the accuracy to 79%. This sets a strong argument in favor of the fact that, for this project, ReLU is superior to Sigmoid. The increment in performance in terms of loss and accuracy suggests that ReLU not only speeds up learning but also helps the network learn better from the data and hence more effectively extract and classify features.

**4.2.3 Third model**

In the third assessment, two major changes were made to the model to boost performance: increasing the number of convolutional filters and switching the optimizer from Stochastic Gradient Descent (SGD) to Adam [1].

The first change involved increasing the number of filters in the convolutional layers. Initially, the model used about 6 filters in the first convolutional layer, but this was increased to 16. Similarly, the second convolutional layer now uses 32 filters, compared to the previous 16. This change allows the network to capture a larger number of features from the input images. The additional filters allow the network to learn more detailed patterns at each layer. For instance, the first convolutional layer captures low-level features like edges and textures, while the deeper layers focus on more abstract patterns such as shapes and facial expressions. By utilizing more filters, the model has a greater ability to detect fine-grained features, which is crucial for complex tasks like distinguishing between different facial expressions. This enhancement also compensates for the fact that the dataset uses grayscale images, which lack color information but still contain valuable variations in pixel intensity.

The second significant change was the transition from SGD to the Adam optimizer. Unlike SGD, which uses a constant learning rate across all weights during training, Adam dynamically adjusts learning rates for each parameter. This dynamic adjustment is particularly useful for deep neural networks, where different layers may benefit from different learning rates. Adam’s ability to prevent the model from getting stuck in local minima is particularly important for tasks with noisy or overlapping data, like facial expression recognition, where variations in lighting, angles, and conditions may make the dataset complex. Adam is known for faster convergence and more reliable performance, which makes it especially suitable for deep learning tasks.

These improvements were expected to enhance the model’s performance by allowing it to capture more complex features and converge faster during training. Indeed, after implementing these changes, the model achieved remarkable results: a loss of 0.0001 and an accuracy of 84.2%. This dramatic improvement in both accuracy and loss confirms that the increased number of filters and the use of the Adam optimizer significantly enhanced the model’s learning process, making it more effective in identifying facial expressions. However, it is important to note that adding complexity to the model can lead to overfitting, especially given the relatively small and simple dataset. Therefore, careful monitoring of training and validation losses is necessary to ensure that the model generalizes well.

**5. Results & Discussion**

The model had three major test runs, in which each had a change introduced to improve performance. Under Sigmoid as the activation function and SGD as the optimizer, an initial model got a loss of 0.6931 and an accuracy of 50% indicating random classification with a poor performance. The vanishing gradient problem was caused by the Sigmoid function preventing weight updates and slowing learning.

However, in the second testing, an introduction of ReLU as an activation function was done to solve the vanishing gradient problems. Because unlike Sigmoid, ReLU allows positive values through and puts negative values to zero which makes it converge quickly. The model has been greatly improved as evidenced by the loss now being reduced to 0.3452 and accuracy increased to 79%. The results confirm that it was a better activation function for this task since a CNN could now extract more meaning rather than using only the bottom features.

The third and final iteration invoked two major improvements: increasing the number of convolutional filters and changing to the Adam optimizer. The change in number was made to increase from 16 to 32 for the first convolutional layer from 6 to utilize 32 instead of 16 as it did for the second convolutional layer. This made it possible for the model to capture more intricate features, thus increasing the ability to differentiate between facial expressions. Also, a change from SGD to Adam meant that the learning rate would now be dynamic, thereby preventing the model from getting stuck in local minima and enabling faster convergence. The performance changes were so outstanding that the model managed to achieve a loss of 0.0001 with an accuracy of 84.2%.

Another important element that this project is concerned with leaving the original infrastructure and getting into a new environment: the original storage intended for primary AWS would soon be replaced as the problem connectivity, later turned to HDFS with more efficient local storage and provided seamless access. TensorFlow was also left for PyTorch because it was not compatible with the computing environment. This involved mainly the absence of AVX support in the virtual machine. Thus, the dynamic computation graph of PyTorch and the broader hardware compatibility provided easier training and less complicated deployment of the models.

However, these improvements made it much better but left it vulnerable to overfitting. So in future work, further regularization techniques like dropout layers or batch normalization methods will be used to minimize overfitting and increase generalization.

**6. Conclusion**

It was possible to come up with a good CNN-based facial expression recognition model, which performed very well with various testing runs. The initial use of Sigmoid and SGD in the model proved ineffective due to the vanishing gradient problem. Use of ReLU improved accuracy remarkably, while increasing the number of filters and switching to Adam as an optimizer hugely increased performance further; hence, it had loss of 0.0001 and accuracy of 84.2%.

Infrastructure changes alongside algorithmic improvements were very important for the success of the project. The change from AWS to HDFS removed the headaches of connection problems. Moreover, changing from TensorFlow to PyTorch brought improvement in compatibility and computational efficiency. These changes highlighted the importance of tool and framework selection in deep learning tasks.

Still such great performanced gains, the issue of overfitting is still present due to high complexity of the model. Future studies should focus on implementing dropout and batch normalization to improve generalization. Further, enlarging the dataset with multi-class classification would prepare the model for wider applications and more realistic scenarios addressing them.

In summary, this study showed that proper architectural decisions as well as the right activation functions and optimizers, can greatly improve CNN performance in facial expression recognition tasks.

**References**

[1] Research on Image Classification Algorithm Based on Pytorch by Lizheng Jiang and Zizhao Zhang 2021 J. Phys.: Conf. Ser. 2010 012009

[2] An Efficient Face Mask Detector with PyTorch and Deep, by Zeelan Basha, Koneru Lakshmaiah Educational Foundation, Guntur, Andhra Pradesh, India.

[3] Analysis of the Application Efficiency of TensorFlow and PyTorch in Convolutional Neural Network by Ovidiu-Constantin Novac, University of Oradea, 410087 Oradea, Romania (2022)

[4] Application research of image classification based on Pytorch and Convolutional neural network by Zhaoyu Li Guangd ong University Of Science And Technology, China (2024)

[5] Deep Residual Learning for Image Recognition by Kaiming He (2015)

[6] Facial Emotion Recognition using CNN in PyTorch by Deyuan Qu, University of North Texas (2023)

[7] Going deeper with convolutions by Christian Szegedy Google Inc. (2014)

[8] Plant image recognition with deep learning: A review by Ying Chen (2023)

[9] Effective machine learning-based skin disease diagnosis using PyTorch by Rohit Kumar, Taiwan (2023)

[10] Dermatologist–level classification of skin cancer with deep neural networks by Andre Esteva (2021)