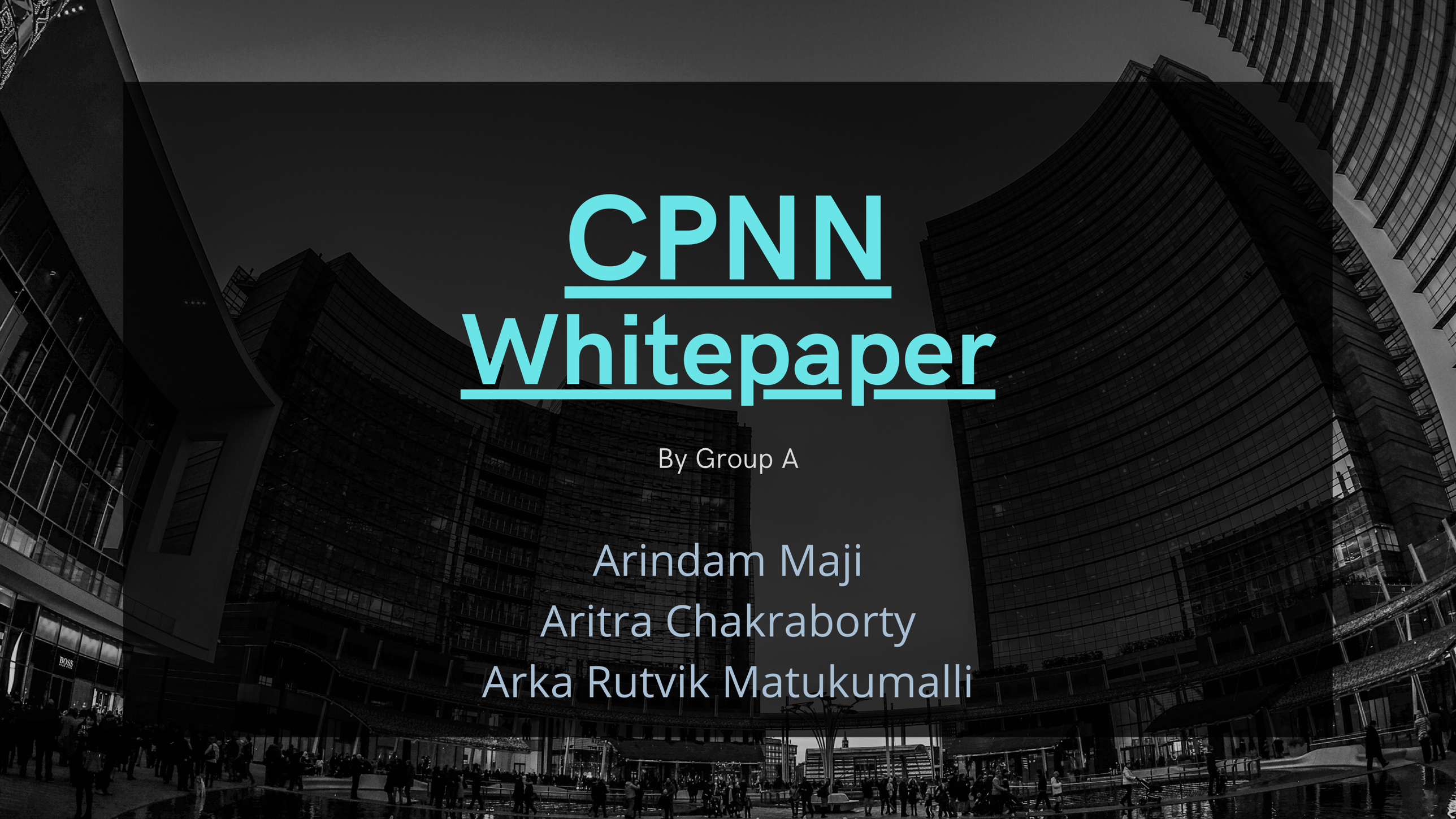


CPNN Whitepaper

By Group A

Arindam Maji
Aritra Chakraborty
Arka Rutvik Matukumalli

Overview

What this whitepaper covers

01 Introduction

02 Background

03 CPNN and its Implementation

04 Discussion

05 Experimental Results

06 Conclusion

Introduction

Face Detection

Robust face recognition could have various applications in our daily life, such as securing restricted areas by using face information, logging into personal accounts without entering any character, and helping catch a person of interest by the police.

Use of Deep Learning

Convolution Neural Network encodes the features obtained from the original input patterns with convolutions and classifies them using the regular machine learning layers.

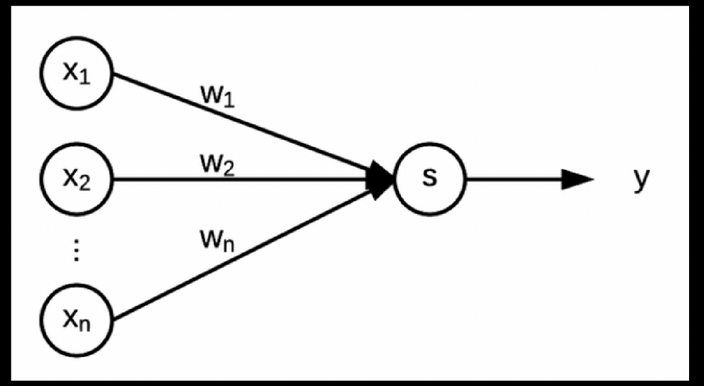
Why CPNN ?

CPNN is a multilayer back propagation neural network based on CNN to learn high dimensional features from a large number of training examples, which are then analyzed by a fully-connected network to perform the recognition.

Background

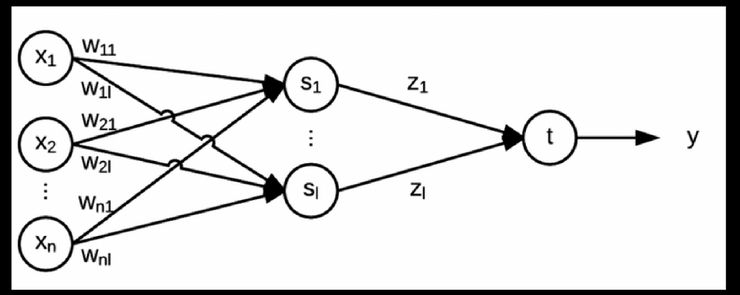
1. Single Layer Perceptron (SLP) :

Perceptron is a supervised learning technology of binary classification. SLP creates linear boundaries to classify the input patterns. Multi-class classification is performed by a "one versus all" strategy. Each input unit is fully- connected with the node(s) in the output layer.



2. Multilayer Perceptron (MLP) :

MLP is an extension of SLP that includes one or more hidden layers in the structure. It creates multiple linear boundaries to assign input data to different groups. Each input unit is connected to the nodes of the first hidden layer, and each node of the first hidden layer is connected to the nodes of the second hidden layer.



Background (cont.)

3. Support Vector Machine (SVM)

SVM creates the maximum margin bound-areas between two different classes. It creates a hyperplane to split the input patterns into two classes.

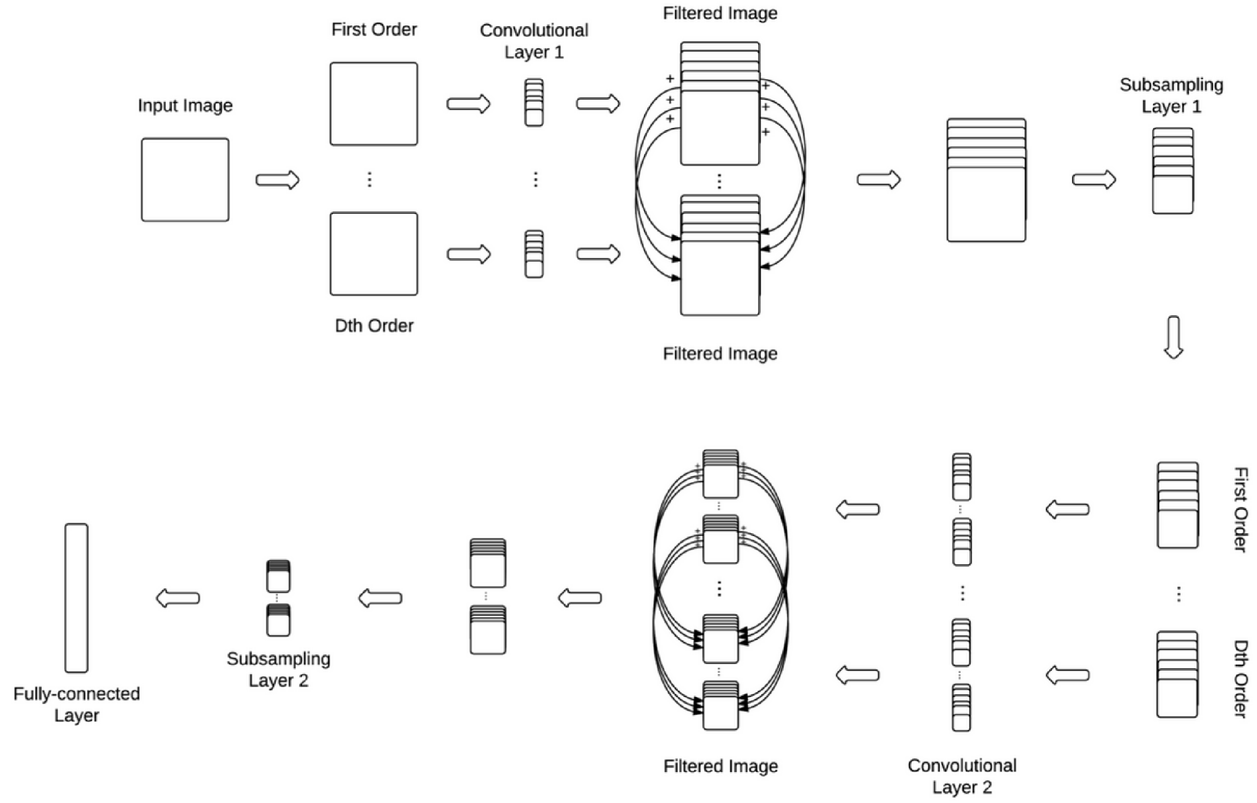
5. Hopfield Network

Hopfield network connects each input unit with a trainable weight matrix to obtain the local minimum of the input data. It is applied to the binary input data (noisy) to remove or reduce noise.

4. Radial Basis Function (RBF)

RBF is a particular neural network that applies a similarity computation in its hidden layers. It creates non-linear boundaries to classify inputs into different classes.

CPNN and its Implementation



Polynomial Expansion

Polynomial expansion is introduced to each convolutional layer, thereby leading to more exact nonlinear fitting of data dependencies and hence better weights could be obtained from the training stage, which in turn facilitated the extraction of the useful information regarding the input image.

Proposed Structure

In the proposed method, the filters between each order are different; in other words, the filters are all independent and self-learned by iteration.

There is no difference between the CNN and CPNN processes for the back propagation of the fully-connected layer. The main differences appear in the convolutional layer in the feed forward and back propagation.

Initialization of CPNN Layer

```
class Convolutional_cpnn(Layer):
    def __init__(self, input_shape, kernel_size, depth):
        input_depth, input_height, input_width = input_shape
        self.depth = depth
        self.input_shape = input_shape
        self.input_depth = input_depth
        self.output_shape = (depth, input_height - kernel_size + 1, input_width - kernel_size + 1)
        self.kernels_shape = (depth, input_depth, kernel_size, kernel_size)
        self.kernels_1 = np.random.randn(*self.kernels_shape)
        self.kernels_2 = np.random.randn(*self.kernels_shape)
        self.kernels_3 = np.random.randn(*self.kernels_shape)
        self.kernels_4 = np.random.randn(*self.kernels_shape)
        self.biases_1 = np.random.randn(*self.output_shape)
        self.biases_2 = np.random.randn(*self.output_shape)
        self.biases_3 = np.random.randn(*self.output_shape)
        self.biases_4 = np.random.randn(*self.output_shape)
```

Forward Pass

$$\mathbf{z}_j = \sum_i (\mathbf{dy}_i * \mathbf{b}_{ij1} + \mathbf{bias}_{2i1} + \mathbf{dy}_i{}^2 * \mathbf{b}_{ij2} + \mathbf{bias}_{2i2} + \ldots + \mathbf{dy}_i{}^D * \mathbf{b}_{ijD} + \mathbf{bias}_{2iD})$$

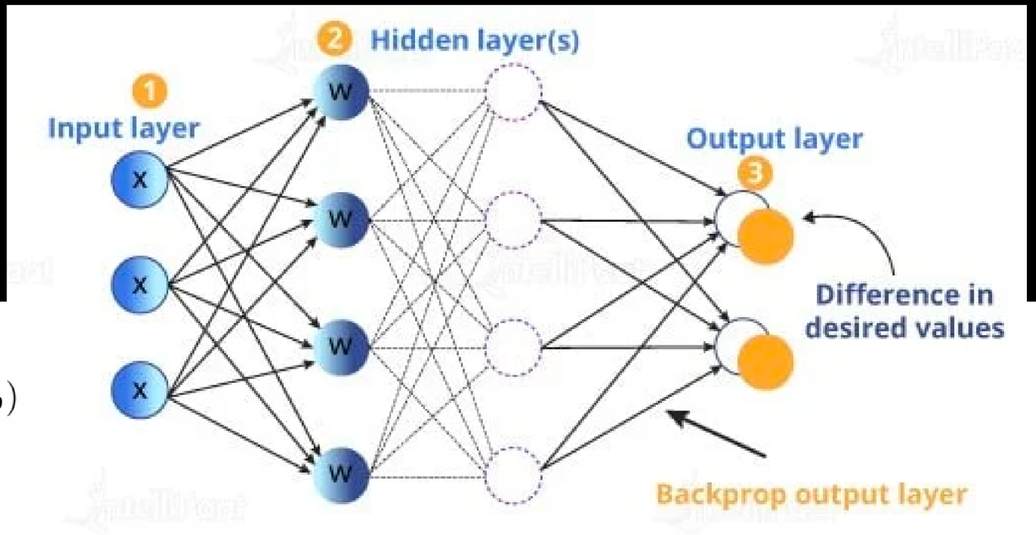
- Input is provided to the first convolutional layer which applies filters to it and its D orders of magnitude.
- This filtered input is given to the sigmoid activation function
- The resultant output is fed to the average pooling layer which downsamples it.
- This process of three steps is again repeated through the second and third convolutional layer.
- The output from these two layers is fed as input to the fully connected layer after the reshaping is done.
- The final output is obtained thus from the dense layer.

Forward propagation

```
def forward(self, input):
    self.input = input
    self.output_1 = np.copy(self.biases_1)
    self.output_2 = np.copy(self.biases_2)
    self.output_3 = np.copy(self.biases_3)
    self.output_4 = np.copy(self.biases_4)
    for i in range(self.depth):
        for j in range(self.input_depth):
            self.output_1[i] += signal.correlate2d(self.input[j], self.kernels_1[i, j], "valid")
    for i in range(self.depth):
        for j in range(self.input_depth):
            self.output_2[i] += signal.correlate2d(self.input[j] * self.input[j], self.kernels_2[i, j], "valid")
    for i in range(self.depth):
        for j in range(self.input_depth):
            self.output_3[i] += signal.correlate2d(self.input[j] * self.input[j] * self.input[j], self.kernels_3[i, j], "valid")
    for i in range(self.depth):
        for j in range(self.input_depth):
            self.output_4[i] += signal.correlate2d(self.input[j] * self.input[j] * self.input[j] * self.input[j], self.kernels_4[i, j], "valid")
    self.output = self.output_1 + self.output_2 + self.output_3 + self.output_4
    return self.output
```

Back Propagation

$$\beta_{dy_i} = \frac{\partial L}{\partial \mathbf{dy}_i} + \frac{\partial L}{\partial \mathbf{dy}_i^2} + \ldots + \frac{\partial L}{\partial \mathbf{dy}_i^D} = \sum_j \boldsymbol{\alpha}_{z_j} * \text{flip}(\mathbf{b}_{ij1}) + \sum_j \boldsymbol{\alpha}_{z_j} * \text{flip}(\mathbf{b}_{ij2}) + \ldots + \sum_j \boldsymbol{\alpha}_{z_j} * \text{flip}(\mathbf{b}_{ijD})$$



- This step is exactly similar to back propagation in fully connected layer of CNN.
- The only difference arises in the back propagation of the convolutional layer since this time we need to adjust all the biases and the filters independent of each other for each order r of the input where r varies from 1 to D orders of magnitude.
- This helps the network to identify non-linear relationship in the input and train the weights accordingly to produce more accurate output.

$$\frac{\partial E}{\partial x_{11}} = \frac{\partial E}{\partial y_{11}} k_{11}$$

$$\frac{\partial E}{\partial x_{12}} = \frac{\partial E}{\partial y_{11}} k_{12} + \frac{\partial E}{\partial y_{12}} k_{11}$$

$$\frac{\partial E}{\partial x_{13}} = \frac{\partial E}{\partial y_{12}} k_{12}$$

$$\frac{\partial E}{\partial x_{21}} = \frac{\partial E}{\partial y_{11}} k_{21} + \frac{\partial E}{\partial y_{21}} k_{11}$$

$$\frac{\partial E}{\partial x_{22}} = \frac{\partial E}{\partial y_{11}} k_{22} + \frac{\partial E}{\partial y_{12}} k_{21} + \frac{\partial E}{\partial y_{21}} k_{12} + \frac{\partial E}{\partial y_{22}} k_{11}$$

$$\frac{\partial E}{\partial x_{23}} = \frac{\partial E}{\partial y_{12}} k_{22} + \frac{\partial E}{\partial y_{22}} k_{12}$$

$$\frac{\partial E}{\partial x_{31}} = \frac{\partial E}{\partial y_{21}} k_{21}$$

$$\frac{\partial E}{\partial x_{32}} = \frac{\partial E}{\partial y_{21}} k_{22} + \frac{\partial E}{\partial y_{22}} k_{21}$$

$$\frac{\partial E}{\partial x_{33}} = \frac{\partial E}{\partial y_{22}} k_{22}$$

$$\begin{cases} y_{11} = b_{11} + k_{11}x_{11} + k_{12}x_{12} + k_{21}x_{21} + k_{22}x_{22} \\ y_{12} = b_{12} + k_{11}x_{12} + k_{12}x_{13} + k_{21}x_{22} + k_{22}x_{23} \\ y_{21} = b_{21} + k_{11}x_{21} + k_{12}x_{22} + k_{21}x_{31} + k_{22}x_{32} \\ y_{22} = b_{22} + k_{11}x_{22} + k_{12}x_{23} + k_{21}x_{32} + k_{22}x_{33} \end{cases}$$

$$\begin{bmatrix} \frac{\partial E}{\partial y_{11}} & \frac{\partial E}{\partial y_{12}} \\ \frac{\partial E}{\partial y_{21}} & \frac{\partial E}{\partial y_{22}} \end{bmatrix} \overset{\star}{full} \begin{bmatrix} k_{22} & k_{21} \\ k_{12} & k_{11} \end{bmatrix}$$

Backward Propagation

```
def backward(self, output_gradient, learning_rate):
    kernels_gradient_1 = np.zeros(self.kernels_shape)
    kernels_gradient_2 = np.zeros(self.kernels_shape)
    kernels_gradient_3 = np.zeros(self.kernels_shape)
    kernels_gradient_4 = np.zeros(self.kernels_shape)
    input_gradient_1 = np.zeros(self.input_shape)
    input_gradient_2 = np.zeros(self.input_shape)
    input_gradient_3 = np.zeros(self.input_shape)
    input_gradient_4 = np.zeros(self.input_shape)

    for i in range(self.depth):
        for j in range(self.input_depth):
            kernels_gradient_1[i, j] += signal.correlate2d(self.input[j], output_gradient[i], "valid")
            input_gradient_1[j] += signal.convolve2d(output_gradient[i], self.kernels_1[i, j], "full")
    for i in range(self.depth):
        for j in range(self.input_depth):
            kernels_gradient_2[i, j] += signal.correlate2d(self.input[j] * self.input[j], output_gradient[i], "valid")
            input_gradient_2[j] += signal.convolve2d(output_gradient[i], self.kernels_2[i, j], "full")
    for i in range(self.depth):
        for j in range(self.input_depth):
            kernels_gradient_3[i, j] += signal.correlate2d(self.input[j] * self.input[j] * self.input[j], output_gradient[i], "valid")
            input_gradient_3[j] += signal.convolve2d(output_gradient[i], self.kernels_3[i, j], "full")
    for i in range(self.depth):
        for j in range(self.input_depth):
            kernels_gradient_4[i, j] += signal.correlate2d(self.input[j] * self.input[j] * self.input[j] * self.input[j], output_gradient[i], "valid")
            input_gradient_4[j] += signal.convolve2d(output_gradient[i], self.kernels_4[i, j], "full")

    self.kernels_1 -= learning_rate * kernels_gradient_1
    self.kernels_2 -= learning_rate * kernels_gradient_2
    self.kernels_3 -= learning_rate * kernels_gradient_3
    self.kernels_4 -= learning_rate * kernels_gradient_4
    self.biases_1 -= learning_rate * output_gradient
    self.biases_2 -= learning_rate * output_gradient
    self.biases_3 -= learning_rate * output_gradient
    self.biases_4 -= learning_rate * output_gradient
    input_gradient = input_gradient_1 + input_gradient_2 + input_gradient_3 + input_gradient_4
    return input_gradient
```

Discussion

Unlike CNN

The proposed method encodes the nonlinear properties to the inputs of each layer. It calculates the relationships between the pixels in each local region, as well as the nonlinearity of each local region. The weights sharing strategy only appears in the same order of each convolutional layer.

Internals

Each unit from each order has inner connections with respective neighbour units. In contrast, the features of the units between different orders are not connected by the filters so that the features obtained by different orders are fused by adding the features together.

Accuracy

This model is quite accurate in predicting the faces of people with a whopping 94% accuracy. It specializes in handling background effects or facial expressions by taking the non-linearity of the input into consideration. Neither does this overfit the data.

Experimental Results

Parameter Evaluation and Results

Parameter Evaluation :

- Number of convolutional and subsampling/pooling layers
- Number of filters in each layer
- Best order number of polynomial inputs

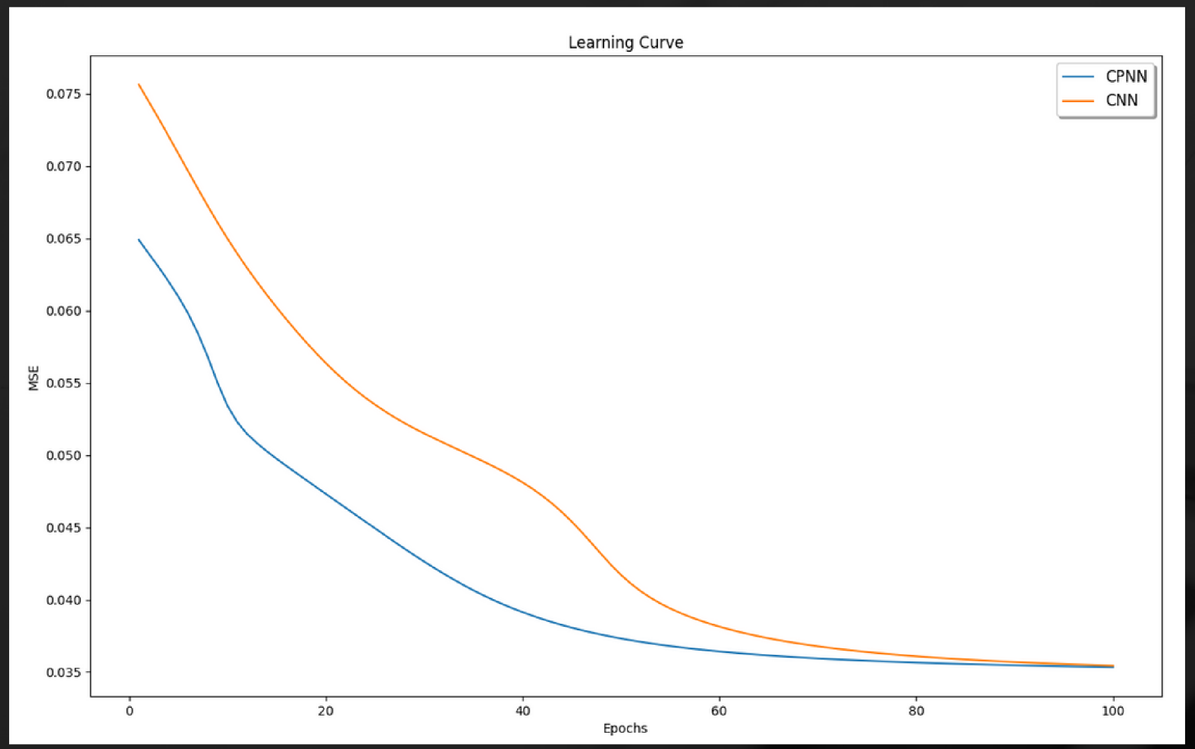
Final results:

- Three layers (convolutional + pooling)
- 12-35-70 filters respectively at each convolutional layer
- Order - 4

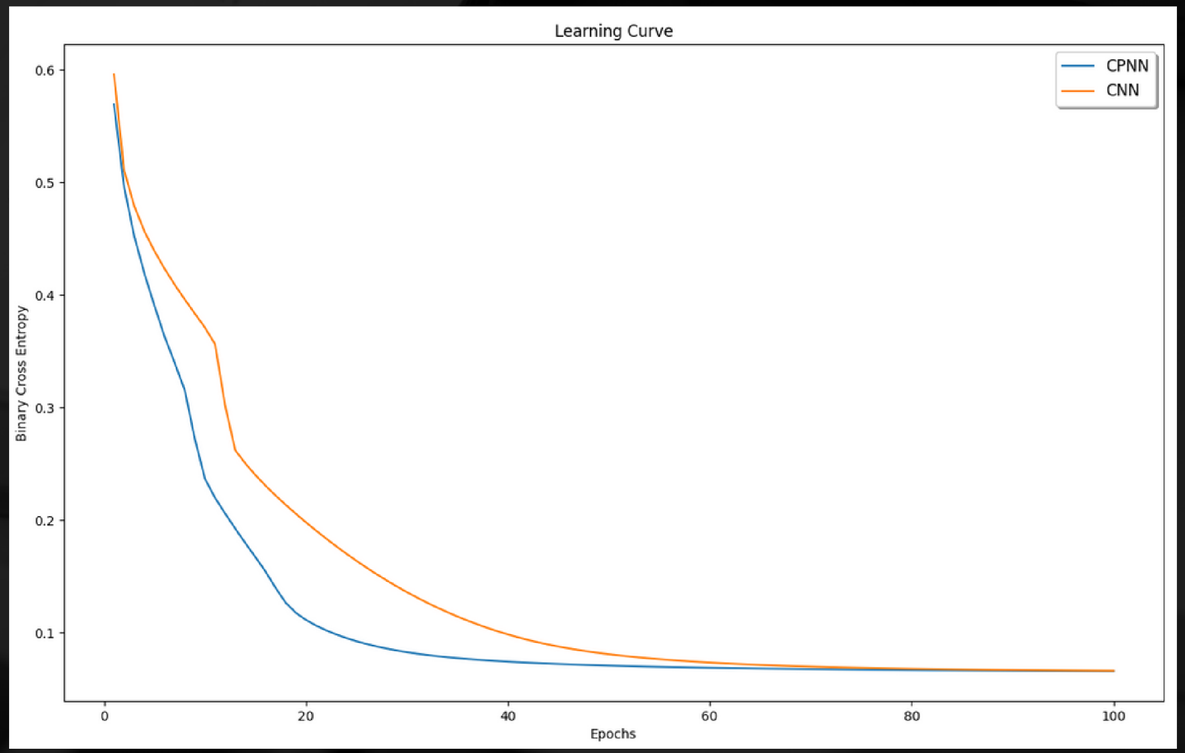
```
# neural network
# cpnn
network1 = [
    Convolutional_cpnn((1, 64, 64), 3, 12),
    Sigmoid(),
    AvgPool((1, 62, 62), 12),
    Convolutional_cpnn((1, 31, 31), 2, 35),
    Sigmoid(),
    AvgPool((1, 30, 30), 35),
    Convolutional_cpnn((1, 15, 15), 2, 70),
    Sigmoid(),
    AvgPool((1, 14, 14), 70),
    Reshape((70, 7, 7), (70 * 7 * 7, 1)),
    Dense(70 * 7 * 7, 100),
    Sigmoid(),
    Dense(100, 10),
    Sigmoid()
]
```

Learning Curves

Mean Squared Error



Binary Cross Entropy Loss



Testing among databases

According to the research paper, the accuracy obtained on different databases :

- Yale(90.89 %)
- CMU AMP (99.95%)
- JAFFE (98.33 %)

According to our testing on the JAFFE database the accuracy obtained was :

Ultimately, when tested in real-world cases the AI model returned a staggering accuracy of 97.22%.

200 images were taken from the JAFFE database and 50 of them were used for training while the rest were used for testing.

Conclusion

- In order to increase the non-linearity and perform a better decision region fitting, the polynomial expansion is introduced to the regular CNN.

- The polynomial expansion extends the input patterns for each convolutional layer with different polynomial terms, therefore the linear curve fitting is modified by the nonlinear curve fitting technique.

- There are many challenges presented by these face images, such as different expressions, lighting conditions, poses, motion blur, etc.

- Despite the challenging image sets, the proposed CPNN still obtains the best accuracy among different methods via different face databases and the images captured in real world environment.

- MSE was used in the research paper but we found out that it is a non-convex function (sometimes we were stuck in a local optima). Instead we used binary cross entropy loss which is a convex function.

Thank you!

Have A Nice Day!

References :

1. https://drive.google.com/file/d/1bBMDiL-Rm_E2HOxnZJzb8cz1KYozqGP_/view?usp=sharing
2. https://youtu.be/Lakz2MoHy6o