A SYSTEM TO CLASSIFY AN OBJECT OF A RGB IMAGE

COURSE NAME : Digital Image Processing

COURSE NUMBER : COSC-6324

COURSE INSTRUCTOR : Minhua Huang

TEAM MEMBERS:

1.Naga Venkata Vasudeva Rao Bodi (A04314290)

2.Sasi Supraja Muthakana (A04314182)

3.Harshita Papaiahgari (A04317038)

4.Sai Avinash Vagicherla(A04312707)

5.Venkata Siva Bharghav Guggilapu(A04324653)

6.Nithin Sampath Rohan Bheemavarapu(A04317338)

CONTENTS

1.Introduction

Convolutional Neural Networks (CNNs) are a specialized class of deep learning models designed for processing structured grid-like data, particularly well-suited for tasks like image analysis. Their impact on computer vision and image recognition has been profound, attributed to their exceptional ability to learn and extract hierarchical representations from visual data. CNNs consist of various layers, including convolutional layers, pooling layers, and fully connected layers, each playing a specific role in feature extraction and classification. Convolutional layers use filters to detect specific features and patterns in input images, while pooling layers reduce spatial dimensions and control overfitting. Fully connected layers leverage the extracted features to classify input images into distinct categories.

For image classification projects, CNNs excel in learning spatial hierarchies of patterns, enabling the identification and differentiation of various objects within images. Their strength lies in automatically learning features without manual extraction, making them highly adaptable to diverse classification tasks and capable of handling complex visual data. This project aims to develop a system for classifying objects in RGB images by harnessing the capabilities of CNNs. The system will learn intricate visual features from input images, undergo training on a dataset of RGB images, fine-tune model parameters, and optimize its architecture to achieve the highest possible classification accuracy.

2. Project Description

2.1 The Filters in Convolutional Neural Networks (CNNs)

In the realm of Convolutional Neural Networks (CNNs), the significance of filters in extracting features from input images cannot be overstated. Filters, alternatively referred to as kernels, are compact matrices employed to extract distinct features like edges, textures, patterns, or shapes from images. Through a process of convolution, these filters are applied to the input image, generating feature maps that emphasize significant patterns within the image.

Why are filters useful?

Feature Extraction: Filters play a crucial role in extracting essential features from the input image, undertaking tasks such as edge detection, blurring, sharpening, and embossing.

Translation Invariance: CNNs achieve translation invariance by employing shared weights across the entire image, allowing them to recognize patterns regardless of their location in the image.

Dimensionality Reduction: The application of filters produces feature maps that contribute to the reduction of image dimensions, enhancing computational efficiency for processing and analysis.

Hierarchical Representation: Filters in various CNN layers capture progressively intricate features, facilitating the learning of hierarchical representations of the input image.

Filter Size and Stride:

The feature extraction process is notably influenced by both the size of the filter and the chosen stride. Opting for a larger filter size enhances the capability to capture intricate patterns, whereas a smaller filter size is more adept at detecting simpler features. The stride, determining the movement step of the filter across the input image, plays a key role. A larger stride leads to smaller feature maps and decreased computational load, while a smaller stride preserves more spatial information.

Examples of Filters:

Gaussian Filter:

Smoothens images by reducing high-frequency noise, often employed in preprocessing.

Box Filter:

Averages pixel values in the filter region, providing a simple form of blurring.

Sobel Filter:

Emphasizes vertical or horizontal edges in an image.

Prewitt Filter:

Similar to the Sobel filter, used for detecting vertical or horizontal edges.

Max Pooling:

Downsamples feature maps, retaining the most significant information for reducing dimensionality.

Min Pooling:

Downsamples feature maps, preserving the minimum values for specific feature information.

2.2 The size of the feature vector obtained from CNN

To determine the size of the feature vector obtained from the CNN model, it is essential to account for the dimensions of the feature maps resulting from the operations at each layer. Considering the structure of the CNN model you supplied, let's calculate the feature vector size. Given an input image size of 32 \times 32 and the utilization of a 2 \times 2 pooling window with a stride of 2 in each max-pooling layer, the computation for the feature vector's size unfolds as follows:

1. Input Image: 32 x 32

2. After max-pooling layer 1: The feature map size would be (32 -2)/2 + 1 = 16 x

16 for each of the 32 channels.

3. After max-pooling layer 2: The feature map size would be (16 - 2)/2 +1 = 8 x 8

for each of the 64 channels.

4. After max-pooling layer 3: The feature map size would be (8 - 2)/2 +1 = 4 x 4

for each of the 128 channels.

5. After max-pooling layer 4: The feature map size would be (4 - 2)/2 +1 = 2 x 2

for each of the 256 channels.

6. After max-pooling layer 5: The feature map size would be (2 - 2)/2 +1 = 1 x 1

for each of the 512 channels.

Therefore, after the final max-pooling operation, the feature map size becomes for each of the 512 channels. This means the resulting feature vector will have a size of 512 \times 1 or simply 512. This represents the size of the feature vector obtained from the CNN model for each input image.

2.3 The number of layers:

1. Convolutional Layers

model.add(layers.Conv2D(64, (3, 3), activation='relu', input_shape=(32, 32, 1))): adds a convolutional layer with 64 filters, each of size 3x3, using ReLU as the activation function.

model.add(layers.Conv2D(128, (3, 3), activation='relu',padding='same')): adds a convolutional layer with 128 filters, each of size 3x3, using ReLU as the activation function.

model.add(layers.Conv2D(256, (3, 3), activation='relu',padding='same')): adds a convolutional layer with 256 filters, each of size 3x3, using ReLU as the activation function.

model.add(layers.Conv2D(512, (3, 3), activation='relu',padding='same')): adds a convolutional layer with 512 filters, each of size 3x3, using ReLU as the activation function.

1. Pooling layers:

model.add(layers.MaxPooling2D((2, 2))): adds a max pooling layer to the neural network model with a 2x2 pooling window, which downsamples the spatial dimensions of the data and helps in creating a more compact and manageable representation.

model.add(layers.MaxPooling2D((2, 2))): adds a max pooling layer to the neural network model with a 2x2 pooling window, which downsamples the spatial dimensions of the data and helps in creating a more compact and manageable representation.

model.add(layers.MaxPooling2D((2, 2))): adds a max pooling layer to the neural network model with a 2x2 pooling window, which downsamples the spatial dimensions of the data and helps in creating a more compact and manageable representation.

model.add(layers.MaxPooling2D((2, 2))): adds a max pooling layer to the neural network model with a 2x2 pooling window, which downsamples the spatial dimensions of the data and helps in creating a more compact and manageable representation.

model.add(layers.MaxPooling2D((2, 2))): adds a max pooling layer to the neural network model with a 2x2 pooling window, which downsamples the spatial dimensions of the data and helps in creating a more compact and manageable representation.

These convolutional and pooling layers operate sequentially to extract features at varying levels of abstraction. Convolutional layers utilize filters to generate feature maps, and subsequent max-pooling layers reduce the size of these maps while preserving essential information. This stepwise process follows a hierarchical approach, allowing the network to grasp intricate representations of the input data, ultimately enhancing its capability in classification or prediction tasks.

1.4 The number full connected layers for classification

model.add(layers.Dense(512, activation='relu')) :

adds a fully connected (dense) layer with 512 units/neurons. Each neuron in this layer is connected to every neuron in the preceding layer.

model.add(layers.Dense(256, activation='relu')):

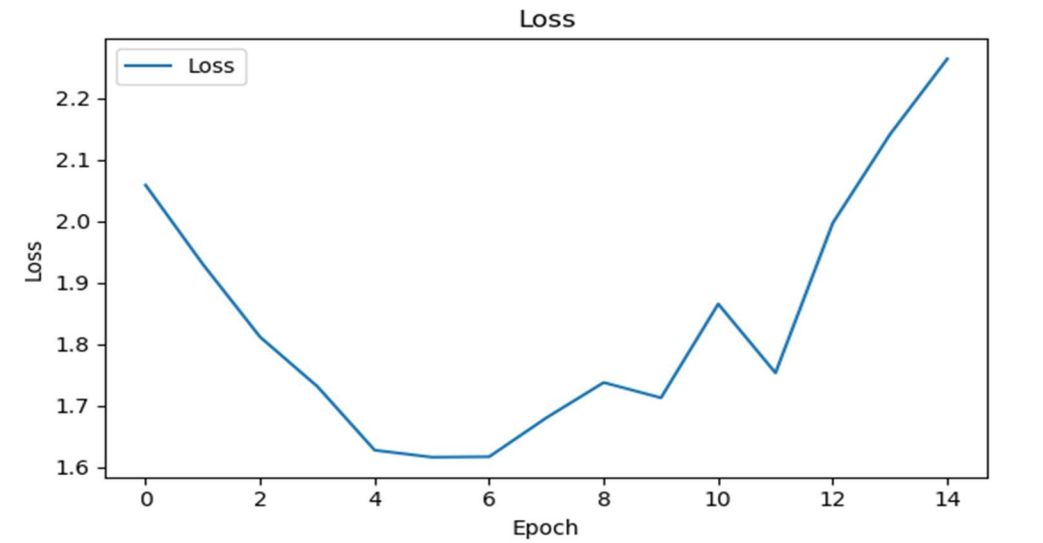
Another fully connected layer is added, the major change being with 256 units.

model.add(layers.Dense(10, activation='softmax')):

This adds the final dense layer with 10 units as we have 10 different classes to classify. The softmax activation function is used in the output layer for multi-class classification. It converts the raw output values into probability distributions over the classes, facilitating the assignment of the input to one of the classes with the highest probability.

1.5 The system loss function and the reason of why we choose it

We have used categorial cross entropy as a system loss function in this system. Categorical Cross-Entropy is a loss function commonly used in machine learning, particularly in the context of multi-class classification problems. It measures the performance of a classification model whose output is a probability distribution over multiple class.



1.6 The activation functions and reasons of why you choose them

Activation functions are pivotal in incorporating non-linear elements into the model. By using non-linear activation functions, the neural network gains the ability to comprehend intricate patterns and dependencies in the data, allowing it to approximate a diverse range of functions. In the context of this CNN model, the selected activation function is Rectified Linear Unit (ReLU), a widely favored choice in deep learning models for its simplicity and effectiveness in addressing the vanishing gradient problem. ReLU is employed following each convolutional and linear layer throughout the network.

Reasons for opting for ReLU activation function are as follows:

Non-linearity: ReLU introduces non-linear characteristics, empowering the network to grasp and represent intricate patterns within the data. This is crucial for effectively training deep neural networks.

Computational efficiency: In comparison to activation functions like sigmoid or tanh, ReLU is computationally efficient. Its simplicity in mathematical operations facilitates faster convergence during the training process.

Sparse activation: ReLU generates sparse activations by nullifying negative values. This sparsity enhances the network's capacity to learn robust and distinctive features, mitigating saturation issues that may arise with alternative activation functions.

1.7 The pooling functions and reasons of why you choose them

In the architecture of a Convolutional Neural Network (CNN), the utilization of max pooling comes after each convolutional layer, serving as a widely adopted method for downsampling and diminishing the spatial dimensions of the feature maps generated through convolutional operations.

The primary motivations for selecting max pooling in the CNN architecture include:

Translation Invariance: Max pooling contributes to the network's ability to recognize essential features, irrespective of their precise location in the input data, enhancing the model's resilience to translations.

Overfitting Reduction: Through the reduction of spatial dimensions in the feature maps, max pooling acts as a preventive measure against overfitting, encouraging the learning of more generalized representations.

Computational Efficiency: Max pooling involves straightforward operations, making it computationally efficient and faster in comparison to alternative pooling techniques.

Feature Invariance: The down sampling achieved by max pooling promotes invariance to minor variations in the input data. These characteristic aids the network in capturing broader patterns and features, contributing to enhanced generalization.

1.8 The Stop Criteria on training procedure

Stop criteria refer to conditions or criteria used to determine when the training of a model should be terminated.

```
model.fit(x_train, y_train_one_hot, epochs=15, batch_size=32, validation_split=0.2)
```

Reasons for choosing 15 ephocs:

Here, the model stops training after 15 epochs is completed. We tried with multiple epochs (30, 40 and 250) and did not see much difference between the accuracy when trained for higher epochs and hence, we used 15 epochs.

Prevent Overfitting: Stop training when the model starts overfitting on the validation set. Efficiency: Save computational resources by stopping when further training may not yield significant improvements.

Efficiency: Save computational resources by stopping when further training may not yield significant improvements.

2. Experiments and Result Discussion

3.1Determining the Learning rate (η):

The learning rate is determined through empirical experimentation. Various values are tested and the learning rate that achieves a balance between convergence speed and stability is chosen. This parameter has a significance role in evaluating a model's performance. Here we have empirically determined an optimal learning rate of 0.001, balancing the convergence speed and stability

3.2 How Long does the training procedure takes and the accuracy of the system

The entire 15 epochs takes 182 minutes to get completed with training accuracy of 78.9% and validation accuracy of 45.10%.

3. Pros and Cons of the system

Pros

Effective Feature Extraction: Our model being a CNN, automatically learns hierarchical representations and features from images without the need for manual feature engineering. This capability is crucial for tasks where the relevant features are complex and multi-level.

Complex Architecture: The structure incorporates multiple convolutional layers, enabling the model to acquire intricate hierarchical features from the input information.

Utilization of Max Pooling: The integration of max pooling layers facilitates the down sampling of feature maps, aiding in the retention of crucial features while reducing spatial dimensions.

Adaptive Average Pooling: The adoption of adaptive average pooling before fully connected layers allows the model to handle input images of varying sizes.

Application of ReLU Activation: The consistent use of the ReLU activation function throughout the network accelerates convergence during training and mitigates challenges associated with the vanishing gradient problem.

Ample Model Capacity: With a substantial number of parameters and layers, the model possesses a considerable capacity to comprehend intricate patterns within the data.

Cons:

Complexity and Overfitting Concerns: The substantial model capacity may result in overfitting, particularly with limited datasets, unless appropriate regularization techniques are applied.

Computational Intensity: The extensive number of parameters and layers can impose significant computational demands, rendering training and inference computationally intensive, especially on devices with constrained resources.

Potential Gradient Issues: Deeper networks may encounter challenges such as vanishing or exploding gradient problems during training, potentially impeding convergence.

Limited Model Interpretability: Due to the intricate architecture, comprehending the internal mechanisms of the model and interpreting its decision-making process can be challenging, thereby limiting interpretability.

Hyperparameter Sensitivity: The architecture's performance may heavily rely on the selection of hyperparameters, including learning rate, batch size, and regularization. Consequently, extensive hyperparameter tuning becomes crucial for achieving optimal results.

4. Further Improvements

Learning Rate Scheduling: Explore various learning rate schedules, such as decay or cyclical learning rates, to identify an optimal strategy for faster convergence and enhanced generalization.

Hyperparameter Tuning: Conduct a thorough search for optimal hyperparameters, encompassing variables like learning rate, batch size, and regularization strength, to attain the highest model performance.

Data Augmentation: Apply techniques for data augmentation to enhance the diversity of the training data, promoting improved generalization of the model.

Regularization: Employ regularization methods like dropout or batch normalization to mitigate overfitting, enhancing the model's capability to generalize effectively to unseen data.

Model Ensemble: Consider creating an ensemble of models to merge multiple predictions, often resulting in improved generalization and overall performance.

Transfer Learning: Leverage pre-trained models, particularly those trained on related tasks or datasets, and fine-tune the network on the specific dataset of interest to leverage the acquired knowledge from the pre-trained model.

Gradient Clipping: Implement gradient clipping as a precautionary measure against gradient explosion or vanishing during training. This can contribute to stabilizing the training process and promoting better convergence.

Advanced Optimization Algorithms: Experiment with diverse optimization algorithms such as Adam, RMSprop, or AdaGrad to enhance convergence speed and overall model performance.