CNN Model

-Vignesh Pitchaiah-

Introduction

Convolutional neural networks (CNNs) are a type of neural network that are commonly used for image classification and recognition tasks. In this report, we describe the architecture of a CNN model that was designed to classify images in a dataset with a size of 128x128 pixels and 3 color channels (RGB) into one of ten classes.

DATA-PREPOSSESSING

I have defined two classes: Data-Prepossess and Model-Trainer.

The Data-Prepossess class loads and prepossessed image data by reading in a CSV file containing the image paths and their corresponding labels. The images are resized to 128x128 pixels and their pixel values are normalized. The prepossessed images and labels are returned as numpy arrays.Then the Model Trainer class builds, trains, and evaluates a convolution neural network (CNN) model using the prepossessed image data and their labels. The CNN model consists of convolution layers, max pooling layers, a dropout layer, a global average pooling layer, and a dense layer. Here I have used Adam optimizer and categorical cross-entropy loss function to compile the model. Then the training history is returned. The accuracy of the model is then evaluated on the testing data. Finally, I have plotted the training and validation loss.

MODEL ARCHITECTURE

Here I have Breakdown of the architecture step wise:
In our data set, we have images with a size of 128x128 pixels and 3 color channels (RGB). The first convolution layer in the architecture takes these images as input and applies 32 filters (or feature maps) of size 3x3 to the input image. Each filter detects a specific pattern or feature in the image, such as edges or textures. The output of this layer is 32 feature maps with the same spatial dimensions as the input image. The output of this layer can be calculated using the formula:

$$output_size = \frac{input_size - filter_size + 2 * padding}{stride} + 1 \tag{1}$$

where input size is the size of the input image (128), filter size is the size of the filter (3), padding is the amount of zero-padding applied to the input (usually set to 1 for a 3x3 filter), and stride is the stride length used to slide the filter over the input (usually set to 1).
So the output size of the first convolution layer would be:

$$output_size = \frac{128 - 3 + 2 * 1}{1} + 1 = 128 \tag{2}$$

Therefore, the output of the first convolution layer would be a feature map of size 128x128x32.

The second convolution layer takes the 32 feature maps generated by the first layer and applies 64 filters of size 3x3 to them. The output size of this layer can be calculated in the same way as the first layer. The ReLU activation function is also applied to introduce non-linearity into the model. So the output size of the second convolution layer would be:

$$output_size = \frac{128 - 3 + 2 * 1}{1} + 1 = 128 \tag{3}$$

Therefore, the output of the second convolution layer would be a feature map of size 128x128x64.

The third layer is a max pooling layer with a pool size of 2x2, which reduces the spatial dimensions of the feature maps by a factor of 2. The output of this layer can be calculated using the formula:

$$output_size = \frac{input_size - pool_size}{stride} + 1 \tag{4}$$

where input size is the size of the input feature map (128), pool size is the size of the pooling window (2), and stride is the stride length used to slide the pooling window over the input (usually set to the pool size). So the output size of the third layer would be:

$$output_size = \frac{128 - 2}{2} + 1 = 64 \tag{5}$$

Therefore, the output of the third layer would be a feature map of size 64x64x64.

The fourth convolution layer takes the 64 feature maps generated by the previous layer and applies 128 filters of size 3x3 to them. The output size of this layer can be calculated in the same way as the previous layers. The ReLU activation function is also applied to introduce non-linearity into the model. So the output size of the fourth convolution layer would be:

$$output_size = \frac{64 - 3 + 2 * 1}{1} + 1 = 64 \tag{6}$$

Therefore, the output of the fourth convolution layer would be a feature map of size 64x64x128.

The fifth layer is another max pooling layer with a pool size of 2x2, which reduces the spatial dimensions of the feature maps by a factor of 2. The output of this layer is a feature map of size 32x32x128. The sixth layer is a flatten layer, which reshapes the output of the previous layer into a 1D array. In this case, the output of the flatten layer is an array of size 131,072 (32x32x128).

The seventh layer is a fully connected layer with 512 neurons, which takes the flattened array as input and produces an output of size 512. The output of this layer is then passed through a ReLU activation function. The eighth layer is a dropout layer with a dropout rate of 0.5, which helps prevent overfitting by randomly dropping out 50 percent of the neurons during training.

The ninth and final layer is another fully connected layer with 10 neurons, which represents the 10 classes of the dataset. This layer uses the softmax activation function to convert the output of the previous layer into a probability distribution over the 10 classes.

The use of max pooling layers and convolutional layers with increasing number of filters helps reduce the spatial dimensions of the feature maps while increasing the number of learned features, allowing the model to learn increasingly complex features and patterns in the input images. The use of dropout and global average pooling layers helps prevent overfitting and reduces the number of parameters in the model, making it more computationally efficient.

The architecture of the CNN can be summarized as follows:

Input image (128x128x3) → convolution layer (32 filters, 3x3) → convolution layer (64 filters, 3x3) → Max pooling layer (2x2) → convolution layer (128 filters, 3x3) → Max pooling layer (2x2) → Dropout layer (0.5) → Fully connected layer (512 neurons) → Dropout layer (0.5) → Fully connected layer (10 neurons with softmax activation) → Output (10 probabilities representing the class probabilities).

Overall, this architecture has 6 convolution layers (including the max pooling layers) and 2 fully connected layers. It has a total of 1,296,778 parameters, which are learned during training using backpropagation and gradient descent.

FINE TUNING

This section I performed a deep learning model trained on a data set of images of different types of bars given. First of the model which I build was trained using transfer learning with the VGG16 architecture, and fine-tuned using the Adam optimizer and then I used it to test it on the Unseen data and evaluated the model's performance using the F1 score, precision, recall, and SME

The pre-trained VGG16 model is loaded with the following architecture:

- Input layer: 128 x 128 x 3 image
- Convolutional layer 1: 64 filters of size 3 x 3 with stride 1 and 'same' padding, followed by ReLU activation function

- Convolutional layer 2: 64 filters of size 3 x 3 with stride 1 and 'same' padding, followed by ReLU activation function
- Max pooling layer 1: Max pooling of size 2 x 2 with stride 2
- Convolutional layer 3: 128 filters of size 3 x 3 with stride 1 and 'same' padding, followed by ReLU activation function
- Convolutional layer 4: 128 filters of size 3 x 3 with stride 1 and 'same' padding, followed by ReLU activation function
- Max pooling layer 2: Max pooling of size 2 x 2 with stride 2
- Convolutional layer 5: 256 filters of size 3 x 3 with stride 1 and 'same' padding, followed by ReLU activation function
- Convolutional layer 6: 256 filters of size 3 x 3 with stride 1 and 'same' padding, followed by ReLU activation function
- Convolutional layer 7: 256 filters of size 3 x 3 with stride 1 and 'same' padding, followed by ReLU activation function
- Max pooling layer 3: Max pooling of size 2 x 2 with stride 2
- Convolutional layer 8: 512 filters of size 3 x 3 with stride 1 and 'same' padding, followed by ReLU activation function
- Convolutional layer 9: 512 filters of size 3 x 3 with stride 1 and 'same' padding, followed by ReLU activation function
- Convolutional layer 10: 512 filters of size 3 x 3 with stride 1 and 'same' padding, followed by ReLU activation function
- Max pooling layer 4: Max pooling of size 2 x 2 with stride 2
- Convolutional layer 11: 512 filters of size 3 x 3 with stride 1 and 'same' padding, followed by ReLU activation function
- Convolutional layer 12: 512 filters of size 3 x 3 with stride 1 and 'same' padding, followed by ReLU activation function
- Convolutional layer 13: 512 filters of size 3 x 3 with stride 1 and 'same' padding, followed by ReLU activation function
- Max pooling layer 5: Max pooling of size 2 x 2 with stride 2
- Fully connected layer 1: 4096 neurons with ReLU activation function
- Fully connected layer 2: 4096 neurons with ReLU activation function

The weights of the model are initialized using the 'imagenet' dataset. All the layers of the pre-trained model are set to non-trainable using the `layer.trainable = False` loop. This means that the weights of these layers will not be updated during training. A new Sequential model is created and the pre-trained VGG16 model is added to it as the first layer with model.add(base model).
A new `GlobalAveragePooling2D()` layer is added to the model, which computes the average pooling of each feature map in the output of the previous layer. The output of the last convolutional layer of the pre-trained VGG16 model has a shape of $4 \times 4 \times 512$. The `GlobalAveragePooling2D()` layer reduces this output to a single vector of length 512 by computing the mean of each feature map. Mathematically, this can be represented as:

$$GAP_i = \frac{1}{n} \sum_{j=1}^{n} x_{ij}$$

where GAP_i is the ith element of the output vector, x_{ij} is the jth element of the ith feature map, and n is the total number of elements in the feature map.

A new `Dense` layer is added to the model with 5 output neurons and a softmax activation function. This layer will produce the final predictions for each input image. Mathematically, the output of the `Dense` layer can be represented as:

$$y_i = \text{softmax}(W_i \cdot x + b_i)$$

where y_i is the ith output neuron, x is the input vector, W_i and b_i are the weight matrix and bias vector of the ith neuron, and `softmax()` is the softmax activation function.
The model is compiled with a categorical cross-entropy loss function, an Adam optimizer, and accuracy as the evaluation metric. Mathematically, the categorical cross-entropy loss function is defined as:

$$L(y_{\text{true}}, y_{\text{pred}}) = -\sum_{i=1}^{m} y_{\text{true},i} \log(y_{\text{pred},i})$$

where y_{true} is the true label vector (one-hot encoded) and y_{pred} is the predicted label vector (output of the `Dense` layer after passing through the softmax activation function).
The Adam optimizer is a variant of stochastic gradient descent (SGD) that uses adaptive learning rates for each weight. The update rule for a weight w_i at time step t is given by:

$$w_i(t+1) = w_i(t) - \frac{\text{learning rate} \cdot m_i(t)}{\sqrt{v_i(t)} + \epsilon}$$

where $m_i(t)$ and $v_i(t)$ are estimates of the first and second moments of the gradient of the loss function with respect to w_i, and ϵ is a small constant to avoid division by zero.
The accuracy metric is defined as the proportion of correctly classified images over the total number of images in the test set.
The model is trained for 1 epoch on the training data using the `fit()` method with the `validation data` argument to evaluate the model performance on the test data after each epoch. During training, the weights of only the `Dense` layer are updated.
The final accuracy is printed using the `evaluate()` method, which returns the loss and accuracy of the model on the test set.

PERFORMANCE EVALUATION

The model's performance was evaluated before and after fine-tuning. Before fine-tuning, the model achieved an accuracy of 85.00 percentage on the validation set after training for 7 epochs. After fine-tuning, the model's accuracy increased to 99.50 percentage on the same validation set, which indicates that the fine-tuning process was successful in improving the model's performance on the specific task it was fine-tuned for. The Confusion Matrix shows that the model is performing very well with high accuracy for all classes, with very few misclassifications. The F1 score of 0.994 and Precision score of 0.995 indicate that the model is correctly predicting the majority of samples across all the classes, while the Recall score of 0.995 suggests that the model is able to identify the majority of the positive samples for all classes. Additionally, the SME score of 0.0025 indicates that the average absolute difference between the predicted and true labels is small, which is a good indicator of model performance. These results suggest that the fine-tuned model is highly accurate and performs well on the specific task it was fine-tuned for.

Table 1: Model performance comparison

	Train data	**Fine-tuned data**
Epochs	1-7	1-2
Loss	1.5799 - 0.5299	0.6804 - 0.4944
Accuracy	0.2587 - 0.7962	0.9925 - 0.9912
Validation Loss	1.4957 - 0.4125	0.5644 - 0.4236
Validation Accuracy	0.35 - 0.85	0.9850 - 0.9950
F1 Score	0.8507101838491373	0.994993843312298
Precision Score	0.8687302431610943	0.9951111111111111
Recall Score	0.85	0.995
SME	0.1025	0.0025

Overall, our model seems to be performing well based on these evaluation metrics.

CONCLUSION

In conclusion, the CNN model architecture used in this report is effective in classifying images based on their features and patterns. The model consists of six convolutional layers, two max pooling layers, two dropout layers, and two fully connected layers, allowing the model to learn increasingly complex features and patterns in the input images. During the finetuning process, the accuracy and loss of the model can be improved, and the confusion matrix is also used to evaluate the true positive and false positive predictions made by the model.