CS584: Machine Learning
Final Project Report

Classification of Doraemon Cartoon Characters from a Custom-Image Dataset Using CNN Model

Santosh Reddy Edulapalle, A20501739, sedulapalle@hawk.iit.edu
Venkata Siva Rupesh Akurati, A20501754, vakurati@hawk.iit.edu
Jack Harrison Mohr, A20503445, jmohr1@hawk.iit.edu

December 3, 2022

Abstract

The goal of this project is to implement a character recognition algorithm on a custom-image dataset of cartoon characters from the cartoon, Doraemon. Our project is inspired by other projects that categorize cartoon characters by image recognition, such as this which is based on the Tom and Jerry cartoon. We implemented a couple of classification models with convolutional neural network and hupertuning them accordingly which resulted in reduction of overfitting and improvement in accuracy.

1 Introduction

Every child's childhood must include cartoons. They are undoubtedly the most well-liked form of children's entertainment, but they also offer much more. Children can learn essential lessons about the world we live in, new emotions, practical challenges in life, and other subjects with the aid of cartoons. On the other hand, recognizing emotions from cartoon characters is still a relatively unexplored field, despite substantial work in this area on real facial images. So this project aims to predict the characters and their emotions from the animated cartoon series Doraemon.

2 Dataset

At the time of starting this project, there was no available dataset consisting of labeled images of Doraemon cartoon characters. Therefore, we created our own dataset. We created all the necessary images for the data from a video that consists of the required cartoon characters. The images are captured at a rate of 1 frame per second using OpenCV. We initially attempted to develop our own cascade classifier to recognize and capture characters images automatically. To build our model, we employed the HAAR cascade fundamentals (figure 1). This method was insufficiently accurate, so we resorted to manual cropping and labeling of images.

However, even after carefully adjusting the model's parameters and photos, the accuracy of the results was not satisfying.

The reason the HAAR model failed is that it relies on features common to photo-realistic human faces. For example, the HAAR cascade operates on the fact that the area around the eyes is darker than the area around the cheeks. Therefore, one of the five HAAR filters catches that eye area.

However, the faces of Doraemon cartoon characters are not consistent with the face features that the HAAR cascade relies on for categorization. For

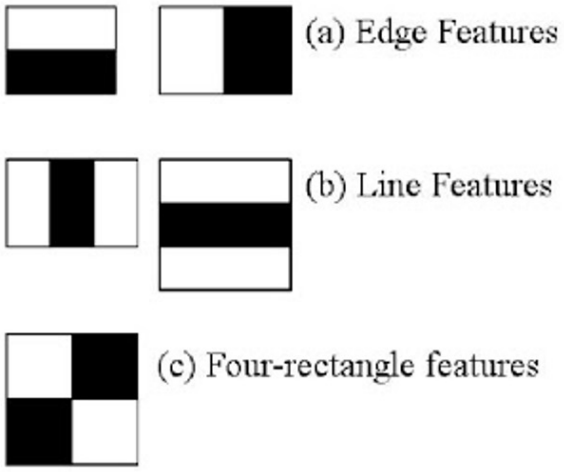

Figure 1: HAAR features

example, everything on the face of the character, Doraemon, is white. Thus there is no differentiation between the cheeks and the eyes in terms of shadow or darkness. This could be one of the factors contributing to model inaccuracy.

Therefore, we manually cropped and labeled the images for this project. We kept those cascade classifier models in .XML documents in our GitHub repository for future use. We initially tried to capture 4 emotions for each of the 3 cartoon characters–Doraemon, Shizuka, Nobita. The emotions are happy, sad, angry, and surprised. Later, based on the time constraints, and availability of data we restricted the project scope to only 2 emotions–Happy and Sad. Hence, we have a total of 6 classes with the details of images as below:

Details of the dataset			
Cartoon Character	**Happy**	**Sad**	**Total**
Doraemon	264	328	597
Nobita	376	300	676
Shizuka	238	155	398
Total	**878**	**783**	**1661**

3 Why CNN?

Convolutional neural networks (CNN) are a subclass of neural networks that are mostly employed in voice and image recognition applications (figure 2). With no loss of information, its integrated convolutional layer lowers the high dimensionality of images. Therefore, CNNs are often used in image recognition systems as they achieve a large decrease in error rate.

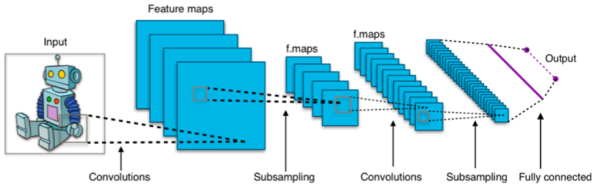

Figure 2: Image explaining CNN process

4 Data Preparation

The very first step of the project is to generate the images from a video. The function ImageSS takes in the path of the video and uses OpenCV to read each frame and captures the images. The function definition can be given as:

As mentioned in the 'dataset' section, we tried to implement the training cascade classifier from scratch and it has been uploaded to Github..

For this, we have to plot -ve annotations (the background images does not contain our character, refer figure 4) and +ve annotations(the characters, refer figure 5).

-ve annotations was relatively easy over +ve annotations, `generate_neg_description_txt_file`. This function runs over the negative images folder and creates a new text file with all the names in the negative image folder + negative tag attached to it.

+ve annotations require manual work. We need to manually annotate the characters frame by frame. But for positives.txt, we need to manually annotate the frame of the face for each image. To help with this, openCV has annotation functions which are only available on version 3x.

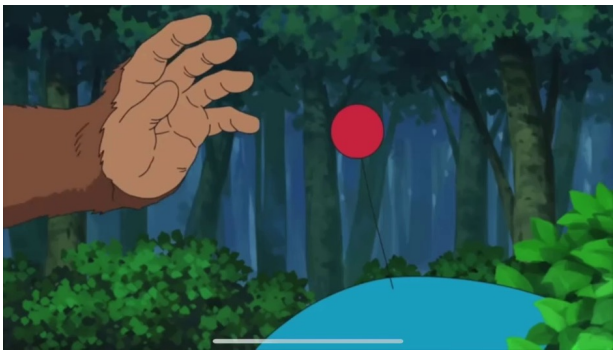

Figure 3: Figure showing -ve annotations

Figure 4: Figure showing +ve annotations

We re-installed openCV 3.4.16 to use its annotation functions. For the rest of the project, we will be using the latest OpenCV version.

The code for creating and generating positive text samples for positive (target characters) is entirely done on CMD. We used `opencv_annotations`, `opencv_createsamples`, `opencv_traincascade` from opencv 3.4.16 version and stored the required models in respective .xml files.

The models we created are: cascadeAll.xml, doremon.xml, nobita.xml, shizuka.xml. i.e., 3 individual models for 3 characters and one model containing multiple characters.

However, even after carefully adjusting the model's parameters and photos, the results' accuracy were not satisfying (figure 5).

Therefore, we manually cropped and labeled the

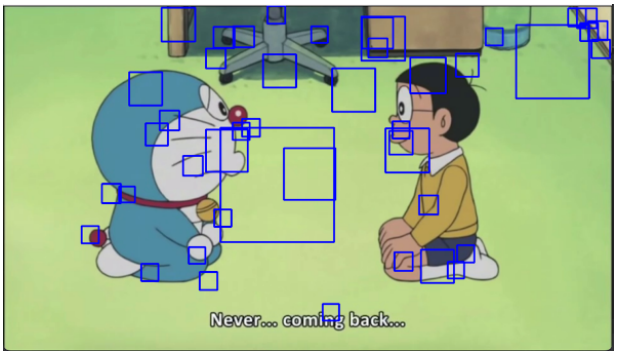

Figure 5: Initial testing results of Cascade Classifier

images for this project. The data has been split in 80-20 fashion to utilize in the model (figure 6).

In our model, each `image_batch` tensor consists of 32 samples with each image of size 256x256x3 where the first two dimensions represent the image height and width, and the third dimension denotes the RGB representation.

The 6 classes created are listed below along with the details of the files:

```
Found 1661 files belonging to 6 classes.
Using 1329 files for training.
Found 1661 files belonging to 6 classes.
Using 332 files for validation.
```

Figure 6: Details of files and thier respective classes

5 Model Building

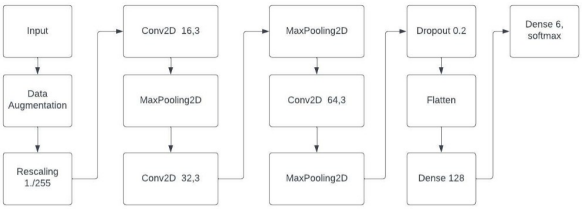

Figure 7: Our CNN model architecture

Here we are using a Sequential model from keras (figure 7). For any Neural Network, it is advisable to scale/normalize the data before giving it as input to the layers. So, here we rescaled the images by dividing all the images with 255(max pixel size).

Here our Keras Sequential model consists of three convolution blocks (tf.keras.layers.Conv2D) with a max pooling layer (tf.keras.layers.MaxPooling2D) in each of them. There's a fully-connected layer (tf.keras.layers.Dense) with 128 neurons (figure 9).

Dense Layer is a simple layer of neurons in which each neuron receives input from all the neurons of the previous layer, thus called as dense. Dense Layer is used to classify images based on output from convolutional layers. A layer contains multiple numbers of such neurons.

Output of kernal filter cannot be directly sent to Dense layer, dense needs 1-D array input. so we use Flatten before Dense. here our final dense layer contains 6 classes.

We used ReLU activation function (refer figure 8).

The Rectified Linear Unit is the most commonly used activation function in deep learning models. The function returns 0 if it receives any negative input, but for any positive value x it returns that value back. So it can be written as: $f(x) = max(0, x)$

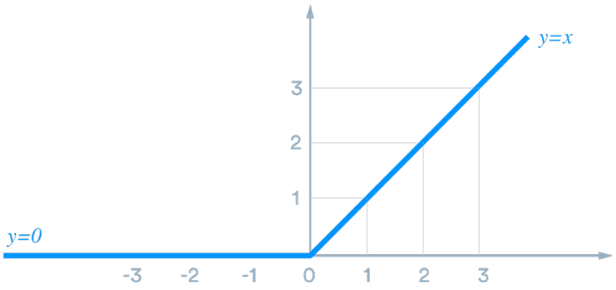

Figure 8: ReLU activation function

Our initial model summary is as below:

From the above summary, we can see the layers used in our Initial CNN model. Parameters (params) are the weights and biases that will be used for computation in all neurons of the CNN.

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
rescaling_2 (Rescaling)      (None, 256, 256, 3)       0

conv2d_3 (Conv2D)            (None, 256, 256, 16)      448

max_pooling2d_3 (MaxPooling  (None, 128, 128, 16)      0
2D)

conv2d_4 (Conv2D)            (None, 128, 128, 32)      4640

max_pooling2d_4 (MaxPooling  (None, 64, 64, 32)        0
2D)

conv2d_5 (Conv2D)            (None, 64, 64, 64)        18496

max_pooling2d_5 (MaxPooling  (None, 32, 32, 64)        0
2D)

flatten_1 (Flatten)          (None, 65536)             0

dense_1 (Dense)              (None, 128)               8388736

dense_2 (Dense)              (None, 6)                 774

=================================================================
Total params: 8,413,094
Trainable params: 8,413,094
Non-trainable params: 0
_________________________________________________________________
```

Figure 9: Initial model summary

6 Model Compilation

For this model, we chose the Adam optimizer from tf.keras.optimizers and the loss function: *tf.keras.losses.SparseCategoricalCrossentropy*.

Adam (Adaptive Moment Estimation) is an algorithm for optimization technique for gradient descent. The method is really efficient when working with large problems involving a lot of data or parameters. It requires less memory and is efficient. Intuitively, it is a combination of the 'gradient descent with momentum' algorithm and the 'RMSP' algorithm.

Adam optimizer involves a combination of two gradient descent methodologies:

1. Momentum

This algorithm is used to accelerate the gradient descent algorithm by taking into consideration the 'exponentially weighted average' of the gradients. Using average makes the algorithm converge towards the minima in a faster pace.

$$w_{t+1} = w_t - \alpha m_t$$

where

$$m_t = \beta m_{t-1} + (1-\beta)\left[\frac{\delta L}{\delta w_t}\right]$$

m_t = aggregate of gradients at time $t[current]$ (initially, $m_t = 0$)
m_{t-1} = aggregate of gradients at time $t-1[previous]$
W_t = weights at time t
$W_t + 1$ = weights at time $t+1$
α_t = learning rate at time t
∂L = derivative of Loss Function
∂W_t = derivative of weights at time t
β = Moving average parameter (const, 0.9)

2. Root Mean Square Propagation (RMSP)
Root mean square prop or RMSprop is an adaptive learning algorithm that rises to improve AdaGrad. Instead of taking the cumulative sum of squared gradients like in AdaGrad, it takes the 'exponential moving average'

$$w_{t+1} = w_t - \frac{\alpha_t}{(v_t+\varepsilon)^{1/2}} * \left[\frac{\delta L}{\delta w_t}\right]$$

where

$$v_t = \beta v_{t-1} + (1-\beta) * \left[\frac{\delta L}{\delta w_t}\right]^2$$

W_t = weights at time t
$W_t + 1$ = weights at time $t+1$
α_t = learning rate at time t
∂L = derivative of Loss Function
∂W_t = derivative of weights at time t
V_t = sum of square of past gradients [i.e., $\text{sum}(\partial L/\partial W_{t-1})$] (initially, $V_t = 0$)
β = Moving average parameter (const, 0.9)
ϵ = A small positive constant (10-8)

Sparse categorical cross entropy: This is a type of loss from *keras.losses*. It is calculated by $-\hat{}ylog(softmax\_output(y))$. We use this cross entropy loss function when there are two or more label classes. The only difference between this and categorical cross entropy is how we provide the labels. Here we provide labels as integers as compared to one-hot representation for the later one.

7 Training the Model

7.1 Initial Run

We trained the model with epochs=10, and the details of loss, accuracy etc. are as below:

Figure 10: Results of initial run

As per the initial results (figure 10), we can see that the validation accuracy has clocked 87% while the training accuracy is almost 100%, and the difference between training results and validation results shows the sign of overfitting, this has been addressed in the further attempts. Initial results were also poor because of improper data for certain characters (Nobita). Most images of this particular character were zoomed in leaving no space for the model to train other features, it led to frequent miscategorizations

of Nobita with Shizuka because of their similar facial features.

Consequently, we made a second attempt at the data preparation, cropping the images more consistently so that each photo showed the entire body instead of just a part of the face (images left- before, right- after, refer figure 11).

Figure 11: Re-cropping of images: Before vs After

Figure 12: Example of data augmentation

7.2 Steps to reduce overfitting

Even after trying the model with new data, our model's predictions were inaccurate because of overfitting. Since we have very little data on certain emotions, our model learned a lot from noises and failed to recognise new data. Thus, we used data augmentation to generate additional training samples. Data augmentation is a process where the new images are created by randomly flipping, rotating, zooming existing images (figure 12).

Another technique to reduce overfitting is to introduce dropout regularization to the network.

When dropout is applied to a layer, it randomly removes some of the layer's output units during training (by setting a neuron's activation to zero). Dropout accepts fractional numbers in the form of 0.1, 0.2, 0.4, etc. as its input value. By doing this, 10%, 20%, or 40% of the output units from the applied layer are dropped at random (figure 13 and 14)

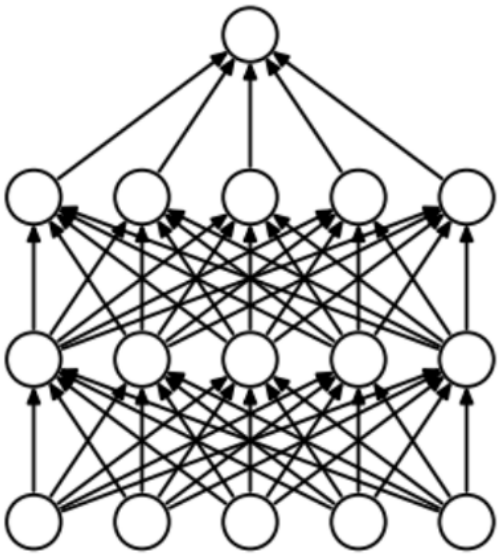

Figure 13: Standard Neural Net

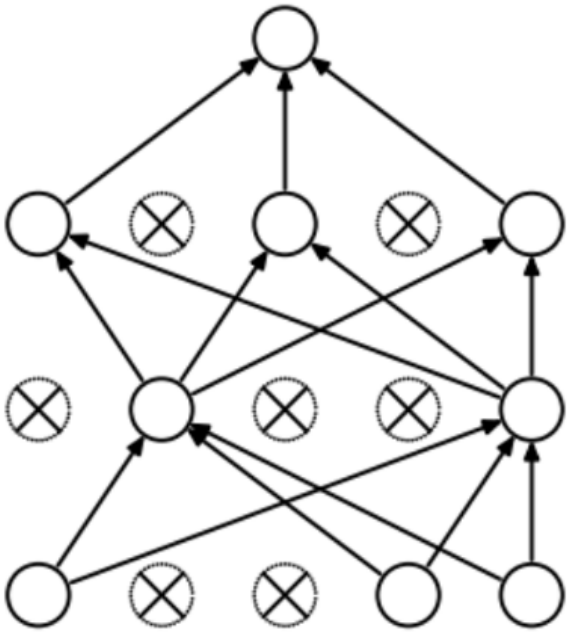

Figure 14: After applying droupout

```
Model: "sequential_3"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 sequential_2 (Sequential)   (None, 256, 256, 3)       0

 rescaling_3 (Rescaling)     (None, 256, 256, 3)       0

 conv2d_6 (Conv2D)           (None, 256, 256, 16)      448

 max_pooling2d_6 (MaxPooling  (None, 128, 128, 16)     0
 2D)

 conv2d_7 (Conv2D)           (None, 128, 128, 32)      4640

 max_pooling2d_7 (MaxPooling  (None, 64, 64, 32)       0
 2D)

 conv2d_8 (Conv2D)           (None, 64, 64, 64)        18496

 max_pooling2d_8 (MaxPooling  (None, 32, 32, 64)       0
 2D)

 dropout_1 (Dropout)         (None, 32, 32, 64)        0

 flatten_2 (Flatten)         (None, 65536)             0

 dense_3 (Dense)             (None, 128)               8388736

 outputs (Dense)             (None, 6)                 774

=================================================================
Total params: 8,413,094
Trainable params: 8,413,094
Non-trainable params: 0
_________________________________________________________________
```

Figure 15: Summary of the new model

7.3 New Model

The model has been enhanced by implementing the changes discussed above and the problem of overfitting is resolved (figure 15).

8 Testing the new model on unseen images

We tested our two models with some new unseen images and the models were able to predict the results with good accuracy (figure 16). One example is posted below (figure 17).

9 Further Enhancement

To further enhance our model, we performed hyperparameter tuning (or hypertuning) to pick the optimal set of hyperparameters. To help optimize the set of hyperparameters, we used Keras Tuner to tune the model parameters. Initially we tried to tune Convolutional layers, but due to hardware resource restrictions, the model was unable to run on full scale. Then we decided to try dropout rate and learning rate tuning.

The results obtained are good but not convincing as they were almost similar to the previous model (`model_6eA`). Hence, for the time being, we are saving the hyperparameter tuning for future development.

Filters and Feature maps:

These are the example samples of our filters (figure 18) and Feature maps outputs (figure 19).

10 Conclusion

Though we achieved a validation accuracy of 87% for the initial model, our model was suffering from

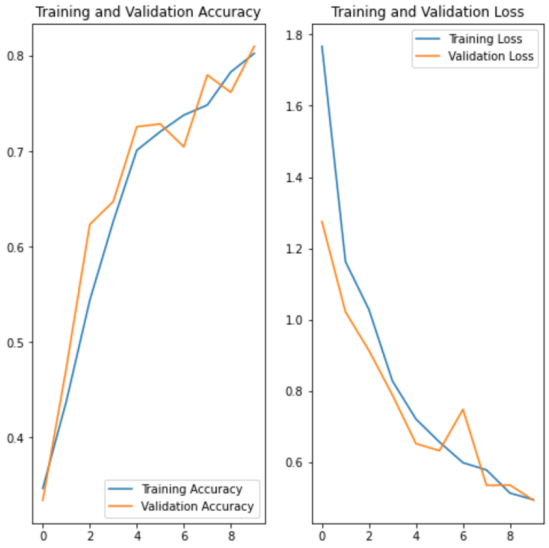

Figure 16: Results of new model showing a drastic reduction in overfitting compared to previous models

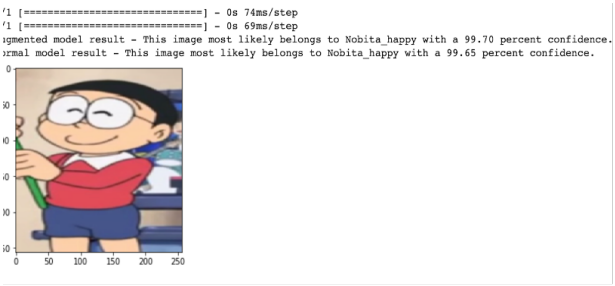

Figure 17: Accurate prediction with 99% confidence

overfitting. Thus we introduced two new layers: a Data augmentation layer and a Dropout layer, which helped drastically reduce the overfitting, but our validation accuracy was around 80%. So, we tried tuning the hyperparameters by tuning dropout rate to between 20% and 50% and the learning rate to between 0.001 and 0.01. Our initial results after hyperparameter tuning were good but not substantially better. We could get almost the same accuracy (79.5%) as the previous model. Thus, we need to do more tuning to get better accuracy, which

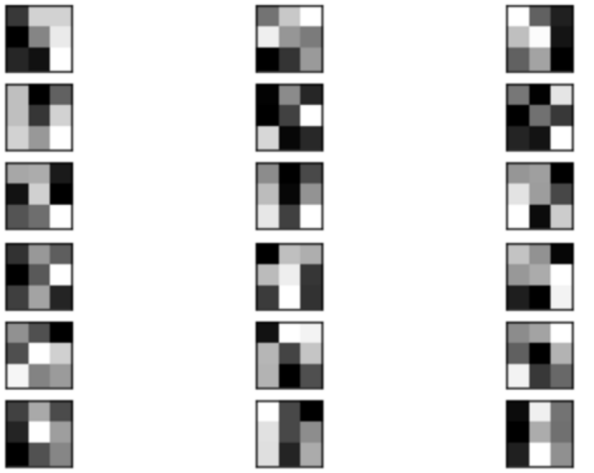

Figure 18: One of the filters used in our CNN model

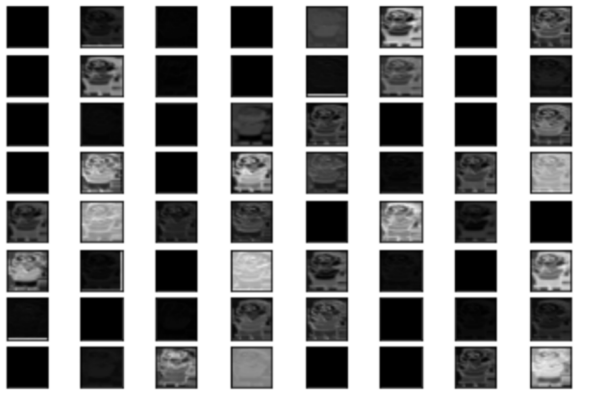

Figure 19: One of the Feature maps output in our model

will be handled in our future development along with our own implementation of a cascade classifier model.

All required files along with the code has been posted on our Github page.

References

[1] N. Lang, "Using convolutional neural network for Image Classification," Medium, 24-Oct-2022

[2] TensorFlow Image classification

[3] Using Convolutional Neural Network for Image Classification

[4] Hayder Hasan et al 2019 IOP Conf. Ser.: Earth Environ. Sci. 357 012035

[5] P. Kalkman, "Increase the accuracy of your CNN by following these 5 tips I learned from the Kaggle Community," Medium, 22-Feb-2021

[6] X. Li, L. Dong, and M. Li, "Improved image classification algorithm based on Convolutional Neural Network," Open Access Library Journal, 01-Dec-2021

[7] Y. amp, C. Aktaş "Fine-tuning a CNN model for Image Classification," Medium, 05-Jan-2022

[8] N Srivastava et al 2014 Journal of Machine Learning Research

[9] B. Baskar, "Tom and jerry image classification," Kaggle, 23-May-2022