University of Pisa

Master degree in Computer Engineering
Course of Intelligent Systems

A CNN-based approach to *CAPTCHA* recognition

Marco Imbelli Cai

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Project Objective

The objective of this project is developing a Neural Network that will be able to recognize five letters text-based CAPTCHAs.

As explained later on, the project will focus on the development of a Neural Network from scratch and the successive improvement of such network, to reach acceptable accuracy levels and reduce as much as possible the phenomenon known as overfitting.

1.1.1 CAPTCHAs

CAPTCHAs, acronym for "Completely Automated and Public Turing test to tell Computers and Humans Apart", are a security mechanism that has been designed to distinguish between human users and automated ones (bots). This technology is widely and commonly used in many websites to prevent automated systems from engaging in abusive activities such as spamming, automatic account creation or data scraping.

CAPTCHAs usually require users to perform tasks that are considered really easy for humans but that are instead difficult for machines, such as identifying distorted sequences of characters (text-based CAPTCHAs), selecting images that match a given description
(image-based CAPTCHAs), or recognizing what is being reproduced in a short audio track (audio-based CAPTCHAs).

By ensuring that only real users can pass such tests, this technology helps protecting websites from automated bots. Examples of text-based CAPTCHAs are shown in Figure 1.1:

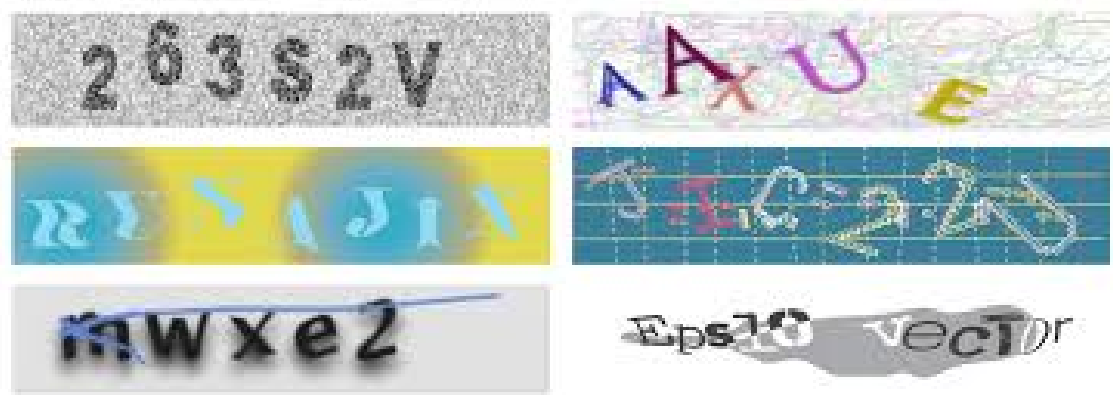

Figure 1.1: Some examples of text-based CAPTCHAs.

Of course, such security mechanisms have been weakened a lot by the increasing popularity of AI techniques able to recognize images and text.

1.1.2 Methodologies Overview

As mentioned before, the type of Neural Network that has been used to attack the problem is the CNN (Convolutional Neural Network). CNNs are a class of deep learning models specifically designed for processing and analyzing structured grid-like data, such as images (despite the name, *text-based CAPTCHAs* are actually images which contain text).

These networks have become the standard for many visual recognition tasks due to their ability to automatically and adaptively learn spatial hierarchies of features directly from input images. Moreover, their capability of learning features that are invariant to translation, scaling and other transformations, make them really suitable for the proposed problem.

The project will focus on two main steps:

- The development of a CNN from scratch and an analysis of its performance
- The improvement of the proposed model to prevent overfitting and increase its performance.

1.1.3 Tools

This project has entirely been developed in the Google Colab environment, a cloud-based platform that allows users to write and execute Python code in an interactive Jupyter notebook environment, supporting powerful ML libraries.

1.1.4 GitHub

The code is freely available and downloadable on GitHub at this link.

Chapter 2

Dataset Analysis

The dataset that has been used for this project is the CAPTCHA Dataset by Parsa Samadnejad, freely available on Kaggle.com (link).

The dataset contains more that 113,000 RGB 5-character CAPTCHA images, and has been generated with the assistance of a CAPTCHA library for PHP.

The images have a resolution of 150×40 pixels, a sample of which is shown in Figure 2.1:

Figure 2.1: A sample of 15 images extracted from the dataset.

The labels can be easily extracted from the images' filenames, and by analyzing such names, we can see that the used characters are 60 in total:

- Digits from 1 to 9 (9 characters)
- Uppercase English alphabet characters (26 characters)
- Lowercase English alphabet characters (25 characters)

The dataset doesn't contain (for some reason) any CAPTCHA with a lowercase 'o' in is characters.

Figure 2.2 clearly shows that characters are evenly distributed in the samples provided by the dataset:

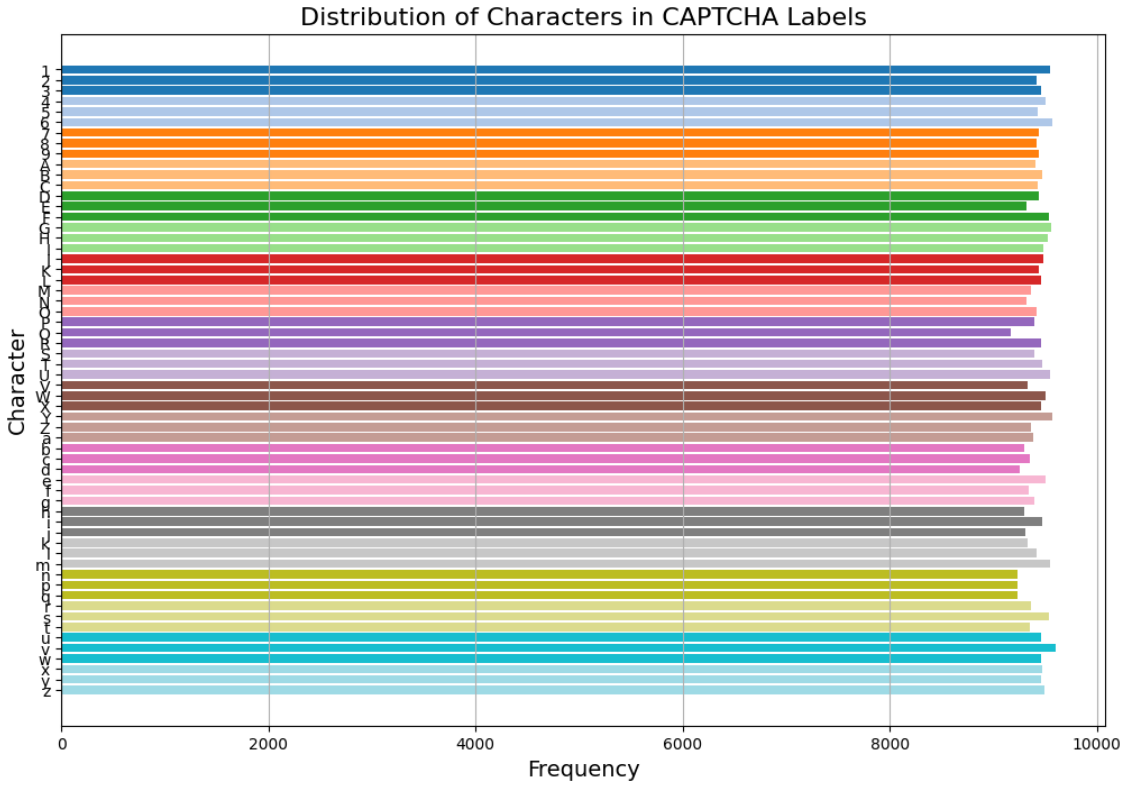

Figure 2.2: Characters distribution in the samples.

Chapter 3

Related Works

In recent years, the application of Convolutional Neural Networks (CNNs) has revolutionized the field of image analysis, offering promising solutions to various problems, including the possible recognition of images resembling CAPTCHAs.

The recognition and classification of the characters of a CAPTCHA image is a complex task due to the intentional noise such images present. Several noteworthy contributions in this area have advanced the capabilities on this matter.

M. Jaderberg, K. Simonyan, A. Zisserman, K. Kavukcuoglu, in their paper *End-to-End Text Recognition with Convolutional Neural Networks* [2], present an approach for recognizing text in images using CNNs. Although it primarily focuses on general text recognition, the methodology is directly applicable to text-based CAPTCHA recognition. The authors propose a fully convolutional approach that can be adapted to CAPTCHA datasets, making it one of the foundational works for CAPTCHA recognition using CNNs.

G. Ye, J. Li, X. Liu in their publication *CAPTCHA Recognition Using Convolutional Neural Networks* [3] specifically target CAPTCHA recognition using a CNN model. The authors design a CNN that effectively handles the distortions and variations present in CAPTCHA images. They also discuss the challenges of overfitting and model robustness in the context of CAPTCHA recognition.

C. Zhang, X. Xiao presented *Breaking Text-Based CAPTCHAs Using COnvolutional Neural Networks* [4], in which they focus on breaking text-based CAPTCHAs using CNNs. The authors experiment with various CAPTCHA datasets and demonstrate that a CNN can be effectively trained to recognize and break these CAPTCHAs with high accuracy.

I. Goodfellow, Y. Bulatov, J. Ibarz, S. Arnoud, V. Shet in their *Breaking Text-Based CAPTCHAs with Deep Learning* [1], discuss how deep learning models, particularly CNNs, can be trained to break text-based CAPTCHAs. The authors experiment with various CNN architectures and demonstrate that these models can achieve high accuracy in solving CAPTCHAs.

This project contributes to the matter by evaluating the effectiveness of CNN models in text-based CAPTCHAs recognition. All the efforts are geared towards the achievement of better security measures to deal optimally with automated bots.

Chapter 4

Methods and Results

In this chapter, we'll discuss the implementation of two Convolutional Neural Network (CNN) models using TensorFlow and Keras, for the purpose of image classification.

The first Neural Network will be built *from scratch*: no regularization techniques will be used. The second architecture will instead be an improvement of the first one, with the objective of reducing overfitting and obtain higher accuracy.

4.1 First Neural Network

4.1.1 Model Architecture

The model architecture is structured as follows:

```
    img = keras.Input(shape=(IMAGE_HEIGHT, IMAGE_WIDTH, CHANNELS))

    conv1 = layers.Conv2D(32, (3, 3), activation='relu')(img)
    mp1 = layers.MaxPooling2D((2, 2))(conv1)

    conv2 = layers.Conv2D(64, (3, 3), activation='relu')(mp1)
    mp2 = layers.MaxPooling2D((2, 2))(conv2)

    flat = layers.Flatten()(mp2)

    dense1 = layers.Dense(128, activation='relu')(flat)

    outputs = []
    for _ in range(CAPTCHA_LENGTH):
        outputs.append(layers.Dense(TOT_CAPTCHA_CHARS, activation='softmax')(dense1))

    model = keras.Model(inputs=img, outputs=outputs)
```

As the code snippet shows, the architecture is rather simple, and a visualization of such can be seen in Figure 4.1.

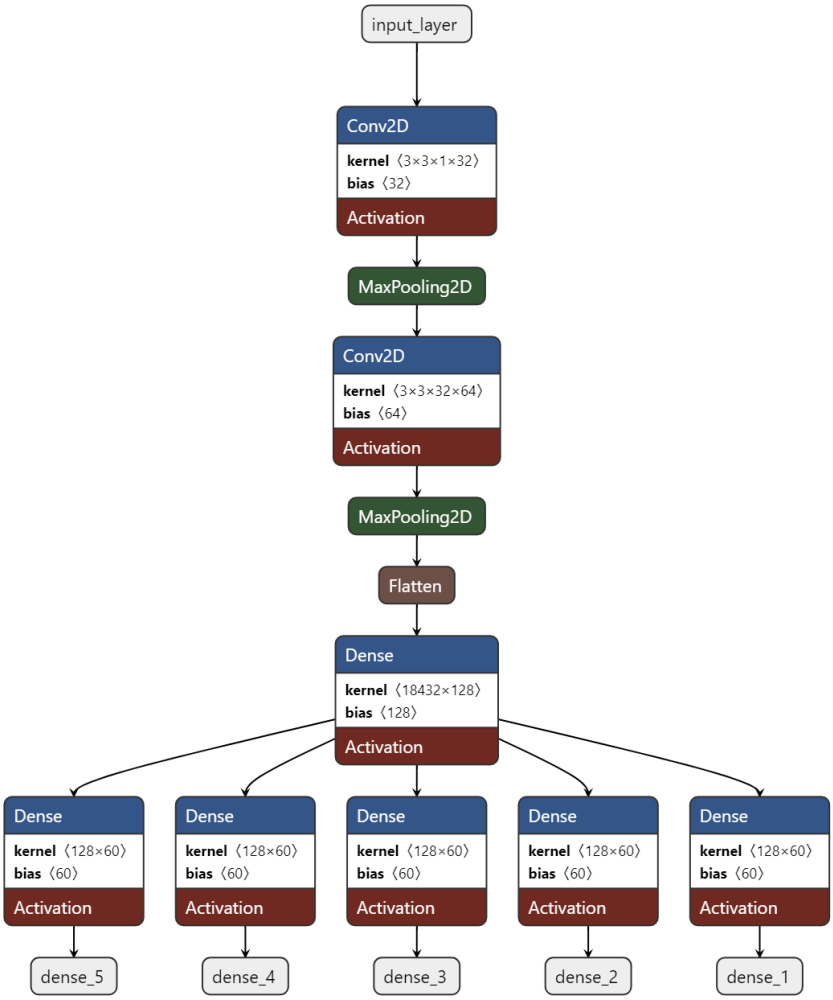

Figure 4.1: Architecture of the simple network

We find an input layer accepting images of shape `IMAGE_HEIGHT x IMAGE_WIDTH x CHANNELS`, i.e., images in the form $(150 \times 40 \times 1)$. The images are fed to the input layer as greyscale images, a sample of which is shown in Figure 4.2.

While RGB images can capture more detailed information, in the case of CAPTCHA recognition, greyscale images are enough to provide all the necessary information about the characters and their structure, while also allowing the NN to be simpler and faster to train (thanks to the usage of just one input channel).

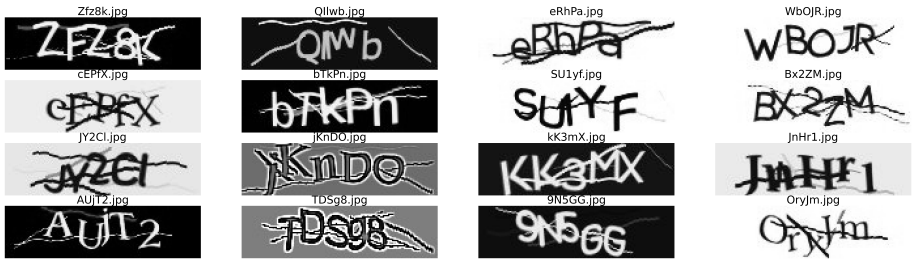

Figure 4.2: A sample of greyscale CAPTCHAs

Multiple convolutional layers with varying number of filters and kernel sizes are introduced to

extract features from the input images. A Max-Pooling layer is added after each Conv2D layer to downsample and normalize the feature maps.

After these layers, a Flatten layer is introduced to convert the output from the convolutional layers into a 1D vector, to prepare the data for the Dense layer.

A fully connected (dense) layer with a *ReLu* activation function is then introduced. The output of such layer will be a 1D vector with 128 elements.

Finally, we can find five output layers. Each one of them employs `TOT_CAPTCHA_CHARS` (i.e. the number of possible characters in the used CAPTCHAs) neurons. A *Softmax* activation function is used here to produce a probability distribution over the possible characters for each output. Each output will be responsible for the prediction of one single character in the multi-character CAPTCHA.

The following is the resulting model (Table 4.1), whose total, trainable and non-trainable parameters are shown in Table 4.2:

Layer (type)	**Output Shape**	**Param #**
input_layer (InputLayer)	(None, 40, 150, 1)	0
conv2d (Conv2D)	(None, 38, 148, 32)	320
max_pooling2d (MaxPooling2D)	(None, 19, 74, 32)	0
conv2d_1 (Conv2D)	(None, 17, 72, 64)	18496
max_pooling2d_1 (MaxPooling2D)	(None, 8, 36, 64)	0
flatten (Flatten)	(None, 18432)	0
dense (Dense)	(None, 128)	2359424
dense_1 (Dense)	(None, 60)	7740
dense_2 (Dense)	(None, 60)	7740
dense_3 (Dense)	(None, 60)	7740
dense_4 (Dense)	(None, 60)	7740
dense_5 (Dense)	(None, 60)	2359424

Table 4.1: From scratch model

Total parameters	**Trainable parameters**	**Non-trainable parameters**
2,416,940 (9.22 MB)	2,416,940 (9.22 MB)	0 (0.00 B)

Table 4.2: Model parameters

4.1.2 Model Training

The model is first compiled and then trained.

Compile Settings

The chosen settings for the compilation phase of the model are the following:

- **Optimizer**: the algorithm used to update the model's weights during training is *Adam*. this has been chosen because of its popularity and adaptive learning rates.
- **Loss function**: *categorical_ crossentropy* has been chosen, since it quantifies the dissimilarity between predicted class probabilities and actual class labels. Its formula is the following: $-\sum_i y_i \cdot log(p_i)$, where y_i is the true probability distribution for class i and p_i is the predicted probability for class i as outputted by the model.
- **Metrics**: the *accuracy* of all the output layers is tracked.

```
    model.compile(
        optimizer='adam',
        loss='categorical_crossentropy',
        metrics=['accuracy'] * CAPTCHA_LENGTH
    )
```

Fitting

The model is trained by employing the `model.fit()` method. Training and validation sets get dynamically split during the training procedure: the training set will be composed of 80% of the samples while the validation one of the remaining 20% (`validation_split`), which is selected on the fly during the fitting procedure.

The training phase spans 30 epochs (`EPOCHS`), with a batch size of 32 (`BATCH_SIZE`).

4.1.3 Training results

The training accuracy of each output layer, together with the validation accuracy for the same output layer has been plotted to evaluate the results of the training phase.

The plots are shown in Figures 4.3 through 4.7.

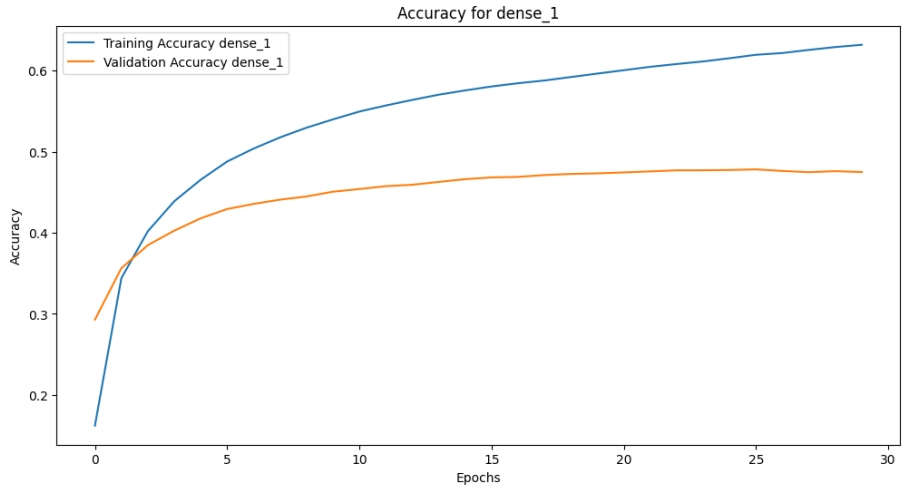

Figure 4.3: Training accuracy and validation accuracy of the first output layer

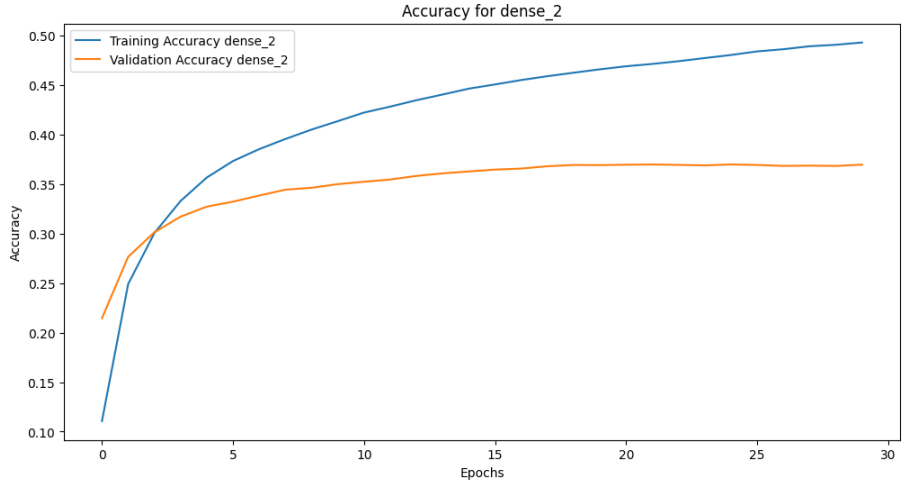

Figure 4.4: Training accuracy and validation accuracy of the second output layer

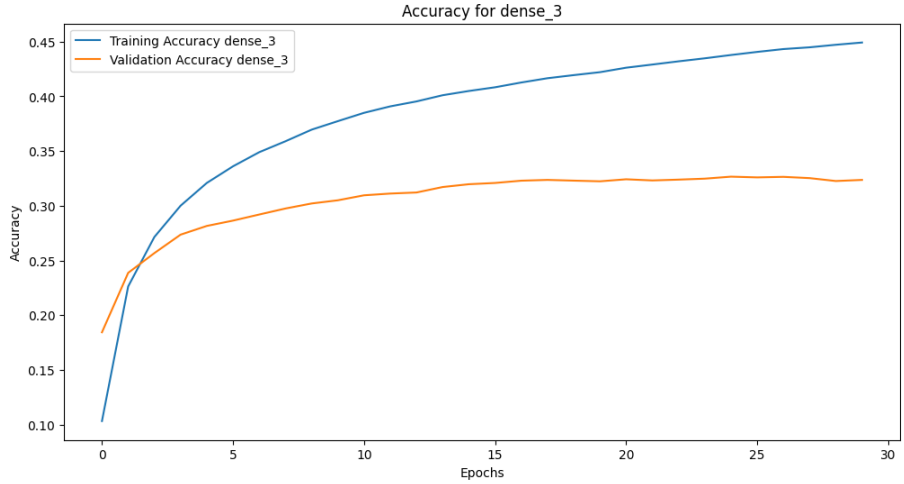

Figure 4.5: Training accuracy and validation accuracy of the third output layer

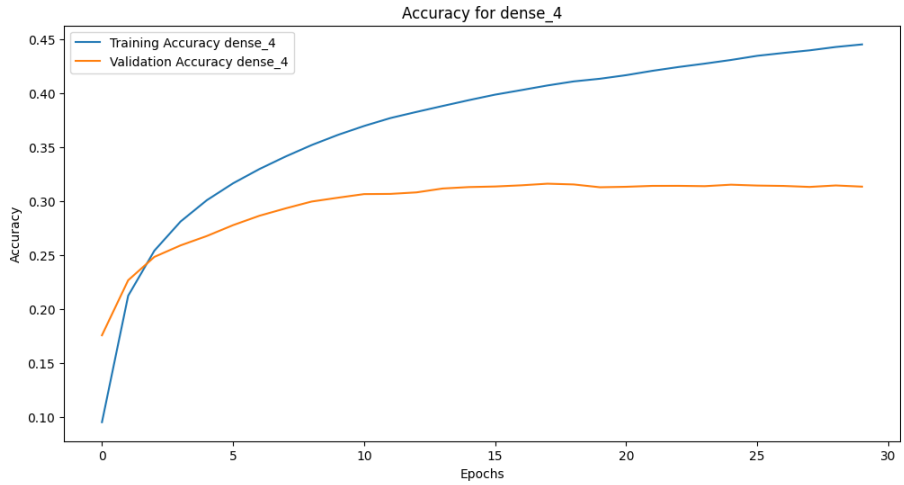

Figure 4.6: Training accuracy and validation accuracy of the fourth output layer

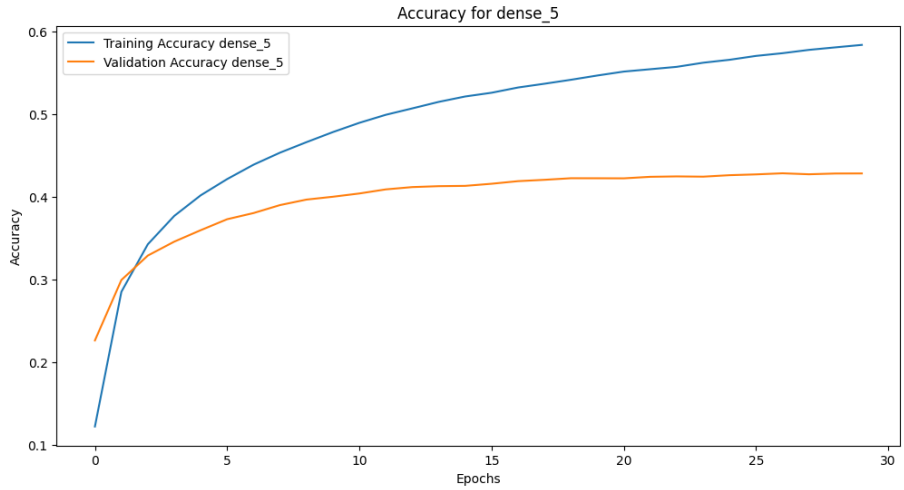

Figure 4.7: Training accuracy and validation accuracy of the fifth output layer

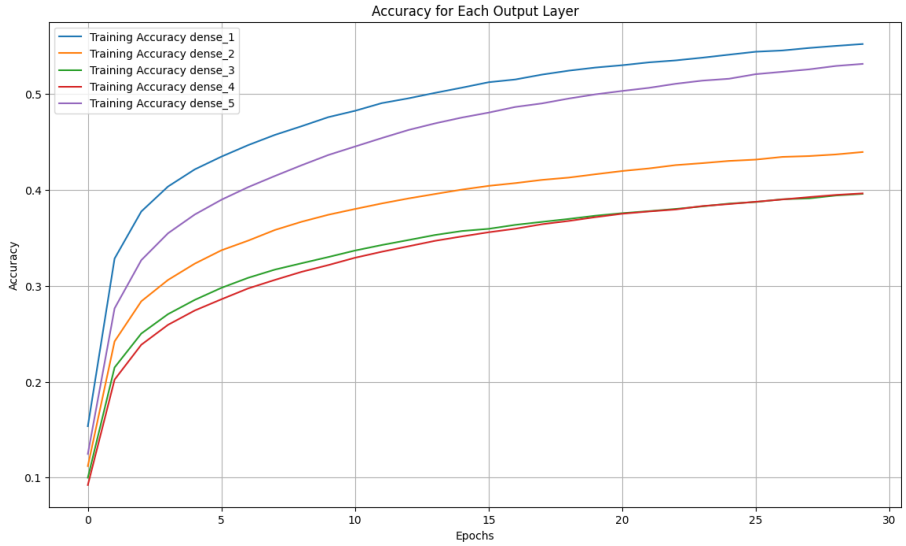

Figure 4.8: Accuracy of all the output layers

The plots show that for all the dense layers, the training accuracy improves as the epochs increase. The validation accuracy initially improves, but after a certain point (around epoch 10-15) starts stagnating.

The gap between these two metrics increases over time, especially after epoch 10. This is a classic sign of overfitting, and it shows that the proposed model performs rather well on the training data, but struggles to maintain high accuracy on the validation samples. By plotting together the training loss and the validation loss, this behavior is better shown (Figure 4.9):

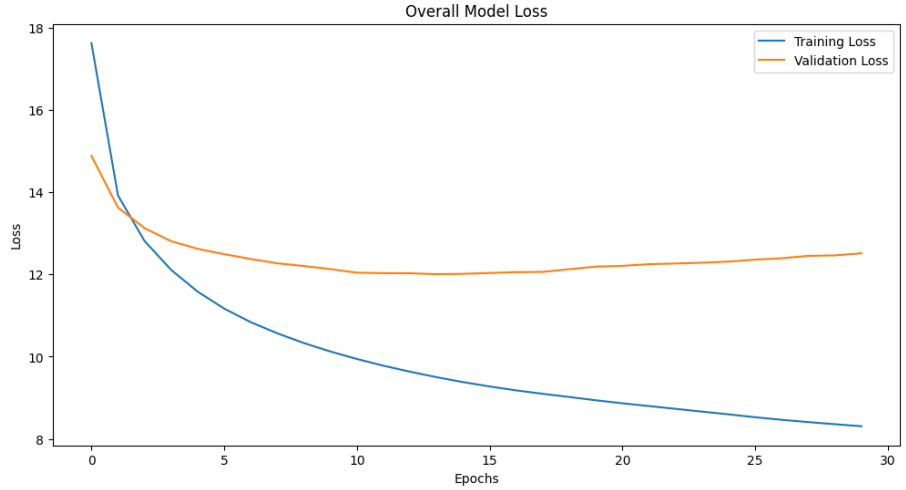

Figure 4.9: Overall model loss

Moreover, since the network has been built from scratch with no notion of regularization methods, the accuracy is still very low. As a matter of fact, Figure 4.8 shows that the reached levels of accuracy are not at all satisfying, with an average of 0.5105 or 51.05%.

The next step is going to be the development of a CNN leveraging methodologies to prevent overfitting. The following section will describe the techniques employed in the second CNN to improve performance and reduce overfitting.

4.2 Improved Neural Network

The model that will be described in this section is obtained by applying regularization and other techniques to the basic CNN discussed previously. With this new network we managed to increase performance and reduce overfitting by a satisfying factor. We'll first discuss the employed techniques and after that the results yielded by the improved network.

The techniques are here listed:

- Regularization techniques
  - Data Augmentation
  - Dropout
  - L2 Regularization
- Callbacks
  - Early Stopping
  - Reduce LR on Plateau

4.2.1 Model Architecture

The architecture of the network is the following:

```
    data_augmentation = keras.Sequential([
        layers.RandomRotation(0.2),
        layers.RandomZoom(0.2),
        layers.RandomTranslation(0.2, 0.2),
        layers.RandomBrightness(0.3),
        layers.RandomContrast(0.3)
    ])

    img = keras.Input(shape=(IMAGE_HEIGHT, IMAGE_WIDTH, CHANNELS))
    augmented = data_augmentation(img)

    conv1 = layers.Conv2D(16, (3, 3), activation='relu', padding='same')(img)
    mp1 = layers.MaxPooling2D(padding='same')(conv1)

    conv2 = layers.Conv2D(32, (3, 3), activation='relu', padding='same')(mp1)
    mp2 = layers.MaxPooling2D(padding='same')(conv2)

    conv3 = layers.Conv2D(64, (3, 3), activation='relu', padding='same')(mp2)
    mp3 = layers.MaxPooling2D(padding='same')(conv3)

    conv4 = layers.Conv2D(128, (3, 3), activation='relu', padding='same')(mp3)
    bn = layers.BatchNormalization()(conv4)
    mp4 = layers.MaxPooling2D(padding='same')(bn)

    flat = layers.Flatten()(mp4)

    outs = []
    for _ in range(CAPTCHA_LENGTH):
        dens1 = layers.Dense(128, activation='relu', kernel_regularizer=keras.
    ↪ regularizers.l2(0.001))(flat)
        drop = layers.Dropout(0.5)(dens1)
        res = layers.Dense(TOT_CAPTCHA_CHARS, activation='softmax')(drop)
        outs.append(res)

    model = keras.Model(inputs=img, outputs=outs)
```

The code snippet shows that the architecture has been updated and increased in complexity with respect to the last network, and a visualization of such can be seen in Figure 4.10.

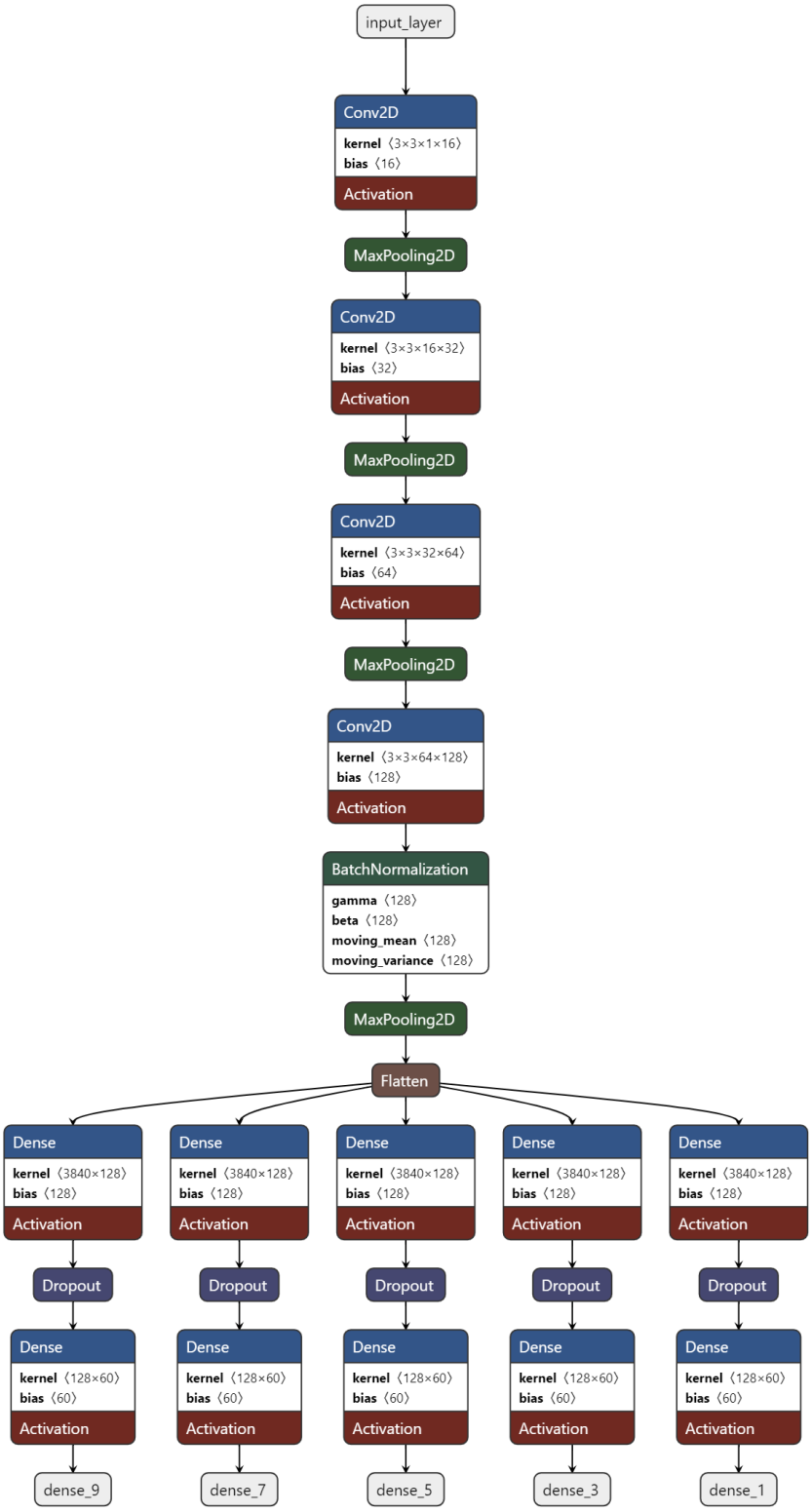

Figure 4.10: Architecture of the updated network

We find an input layer accepting images of shape `IMAGE_HEIGHT x IMAGE_WIDTH x CHANNELS`, i.e., images in the form $(150 \times 40 \times 1)$. The

images are fed to the input layer as greyscale images, as it was done for the first network.

Multiple convolutional layers with varying number of filters are introduced to extract features from the input images. A Max-Pooling layer is added after each Conv2D layer to downsample and normalize the feature maps.

A Batch normalization layer has then been added after the convolutional steps to help stabilizing and accelerate the training of the network.

After these layers, a Flatten layer is introduced to convert the output from the convolutional layers into a 1D vector, to prepare the data for the Dense layer.

The network then branches in five paths, each one consisting of a first dense layer (128 neurons, *ReLU* activation function), a dropout layer and finally a second dense layer (`TOT_CAPTCHA_CHARS` neurons, *Softmax* activation function). The output of each path is then connected to separate output nodes.

The following is the resulting model (Table 4.3), whose total, trainable and non-trainable parameters are shown in Table 4.4:

Layer (type)	**Output Shape**	**Param #**
input_layer (InputLayer)	(None, 40, 150, 1)	0
conv2d (Conv2D)	(None, 40, 150, 16)	160
max_pooling2d (MaxPooling2D)	(None, 20, 75, 16)	0
conv2d_1 (Conv2D)	(None, 20, 75, 32)	4640
max_pooling2d_1 (MaxPooling2D)	(None, 10, 38, 32)	0
conv2d_2 (Conv2D)	(None, 10, 38, 64)	18496
max_pooling2d_2 (MaxPooling2D)	(None, 5, 19, 64)	0
conv2d_3 (Conv2D)	(None, 5, 19, 128)	73856
batch_normalization (BatchNormalization)	(None, 5, 19, 128)	512
max_pooling2d_3 (MaxPooling2D)	(None, 3, 10, 128)	0
flatten (Flatten)	(None, 3840)	0
dense (Dense)	(None, 128)	491648
dense_2 (Dense)	(None, 128)	491648
dense_4 (Dense)	(None, 128)	491648
dense_6 (Dense)	(None, 128)	491648
dense_8 (Dense)	(None, 128)	491648
dropout (Dropout)	(None, 128)	0
dropout_1 (Dropout)	(None, 128)	0
dropout_2 (Dropout)	(None, 128)	0
dropout_3 (Dropout)	(None, 128)	0
dropout_4 (Dropout)	(None, 128)	0
dense_1 (Dense)	(None, 128)	7740
dense_3 (Dense)	(None, 128)	7740
dense_5 (Dense)	(None, 128)	7740
dense_7 (Dense)	(None, 128)	7740
dense_9 (Dense)	(None, 128)	7740

Table 4.3: Improved model

Total parameters	**Trainable parameters**	**Non-trainable parameters**
2,594,604 (9.90 MB)	2,594,348 (9.90 MB)	256 (1.00 KB)

Table 4.4: Improved model parameters

4.2.2 Regularization techniques

Data augmentation

Data augmentation is crucial in this setting to improve the robustness and generalization skills of the CNN. This technique is used here to introduce variability during training, which then forces the model to learn more robust features rather than overfitting the specific training images.

The employed augmentation layer is the following:

```
data_augmentation = keras.Sequential([
    layers.RandomRotation(0.2),          # Randomly rotate within 20%
    layers.RandomZoom(0.2),              # Randomly zoom within 20%
    layers.RandomTranslation(0.2, 0.2),  # Randomly translate within 20% of width/
↪ height
    layers.RandomBrightness(0.3),        # Randomly change brightness within +/- 30%
    layers.RandomContrast(0.3)           # Randomly change contrast within +/- 30%
])
```

The choice of augmentation techniques was driven by the images present in the used dataset, and are here described:

- Random Rotation (0.2): rotations simulate various angles a text might appear. CAPTCHA systems often vary the orientation of characters to make automated recognition difficult. This augmentation ensures that the model is invariant to small rotations.
- Random Zoom (0.2): zooming helps simulating the effect of characters being closer or farther from the camera, or differently scaled in the CAPTCHA. This makes the model robust to variations in character size, which can occur due to different font sizes or spacing in CAPTCHAs.
- Random Translation (0.2): translating the image randomly in width and height simulates characters being offset or misaligned in the CAPTCHA image. This helps the model generalize to CAPTCHAs where characters are not perfectly centered, ensuring it can still recognize them despite positional changes.
- Random Brightness (0.3): variations in brightness can occur due to different lighting conditions or noise in the CAPTCHA generation process. This augmentation ensures that the CNN can handle variations in image brightness, which is common in CAPTCHAs that are designed to confuse OCR systems by adding noise or varying the background.
- Random Contrast (0.3): changing contrast simulates different levels of distinction between the text and the background in the CAPTCHA. This helps the model become more robust to CAPTCHAs with poor contrast, where distinguishing the text from the background might be challenging.

Dropout

The use of dropout primarily focuses on preventing overfitting and enhancing the generalization ability of the model. Dropout is applied after the first dense layers and before the final classification layers: this is done because of the large number of parameters of the fully connected layers, which make them more prone to overfitting.

```
drop = layers.Dropout(0.5)(dens1)
```

A rather aggressive dropout rate of 50% has been chosen, to ensure enough neurons remain active to learn CAPTCHA distortions while still providing strong regularization.

L2 Regularization

L2 regularization is employed to prevent overfitting by adding a loss penalty to the loss function: this discourages the model from learning too complex patterns that might not generalize well unseen data.

```
    dens1 = layers.Dense(
        128,
        activation='relu',
        kernel_regularizer=keras.regularizers.l2(0.001)
    )(flat)
```

Since CAPTCHAs can be designed to be complex, there's a risk of overfitting to the training data, especially with a deep network. L2 regularization helps mitigate this risk by penalizing large weights.

4.2.3 Callbacks

Two callbacks are employed to manage the training process effectively.

Early Stopping

Early stopping is used to prevent overfitting by halting the training process when the model's performance on the validation set starts to degrade.

```
    early_stopping = EarlyStopping(
        monitor='val_loss',
        patience=15,
        min_delta=0.001,
        mode='min',
        verbose=1,
        restore_best_weights=True
    )
```

The validation loss (`monitor='val_loss'`) gets monitored with a patience of 15 epochs (`patience=15`): the training will be stopped if no improvement is seen in validation loss for 15 consecutive epochs. The minimum change in the monitored quantity to qualify as an improvement is set to 0.001 (`min_delta=0.001`): if the change in validation loss is less than this value, it is not considered an improvement.

After stopping, the model will restore the weights from the epoch with the best (i.e. lowest) validation loss (`restore_best_weights=True`).

Reduce LR on Plateau

The ReduceLROnPlateau callback reduces the learning rate when a metric has stopped improving, which can help in fine-tuning the model and converging to a better solution. It helps in adjusting the learning rate dynamically based on the performance.

```
    lr_schedule = keras.callbacks.ReduceLROnPlateau(
        monitor='val_loss',
        factor=0.5,
        patience=5,
        min_lr=1e-6,
        verbose=1
    )
```

The learning rate will be reduced based on changes in the validation loss (`monitor='val_loss'`) by a factor of 0.5 (`factor=0.5`), and will not be reduced below $1e-6$ (`min_lr=1e-06`). The number of epochs to wait for an improvement in validation loss before reducing the learning rate is set to 5 (`patience=5`).

4.2.4 Model Training

The model is first compiled and then trained.

Compile Settings

The chosen settings for the compilation phase of the model are the following:

- **Optimizer**: the algorithm used to update the model's weights during training is *Adam* with a learning rate of $1e-4$. Such learning rate might be reduced during the training procedure by the *ReduceLROnPlateau* callback.
- **Loss function**: *categorical_ crossentropy* has been chosen, since it quantifies the dissimilarity between predicted class probabilities and actual class labels.
- **Metrics**: the *accuracy* metric is used, and is tracked for all the output layers.

```
model.compile(
    optimizer=keras.optimizers.Adam(learning_rate=1e-4),
    loss=['categorical_crossentropy'] * CAPTCHA_LENGTH,
    metrics=['accuracy'] * CAPTCHA_LENGTH
)
```

Fitting

The model is trained by employing the `model.fit()` method. Training and validation sets have been previously split during the preprocessing stage: the training set will be composed of 70% of the samples while the validation one of 20%. The test set will be composed of the remaining 10%. The training phase spans 100 epochs (`EPOCHS`, possibly less due to the early stopping callback) with a batch size of 64 (`BATCH_SIZE`).

4.2.5 Training results

The training accuracy of each output layer, together with the validation accuracy for the same output layer has been plotted to evaluate the results of the training phase. The plots are shown in Figures 4.11 through 4.15.

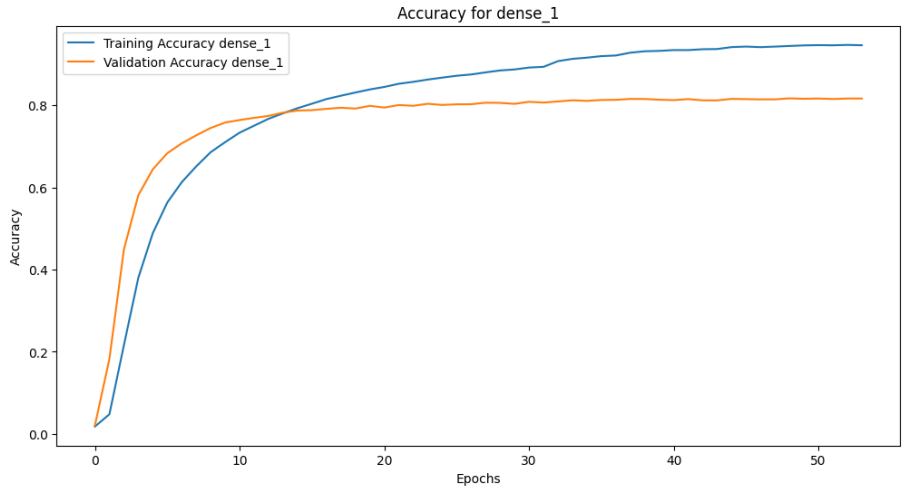

Figure 4.11: Training accuracy and validation accuracy of the first output layer

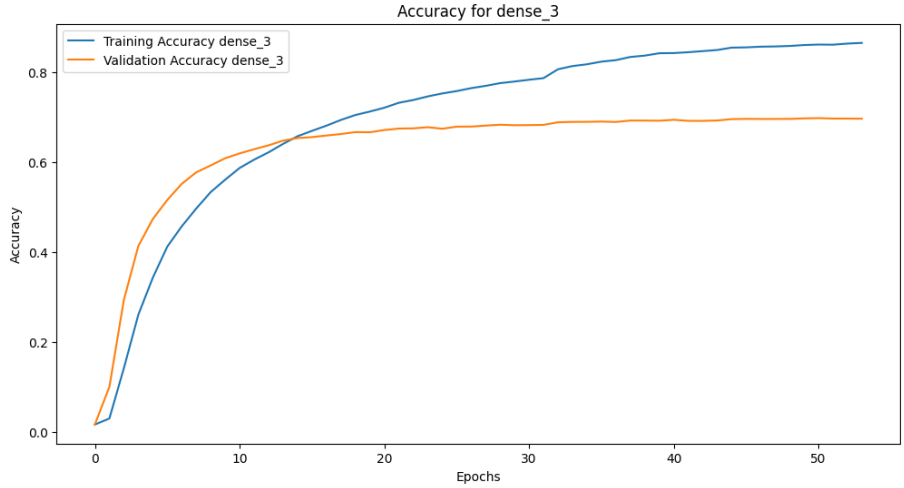

Figure 4.12: Training accuracy and validation accuracy of the second output layer

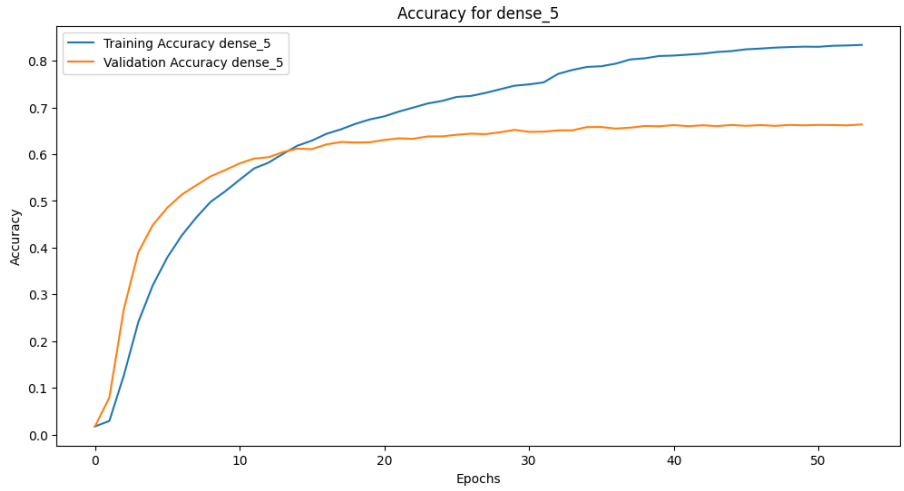

Figure 4.13: Training accuracy and validation accuracy of the third output layer

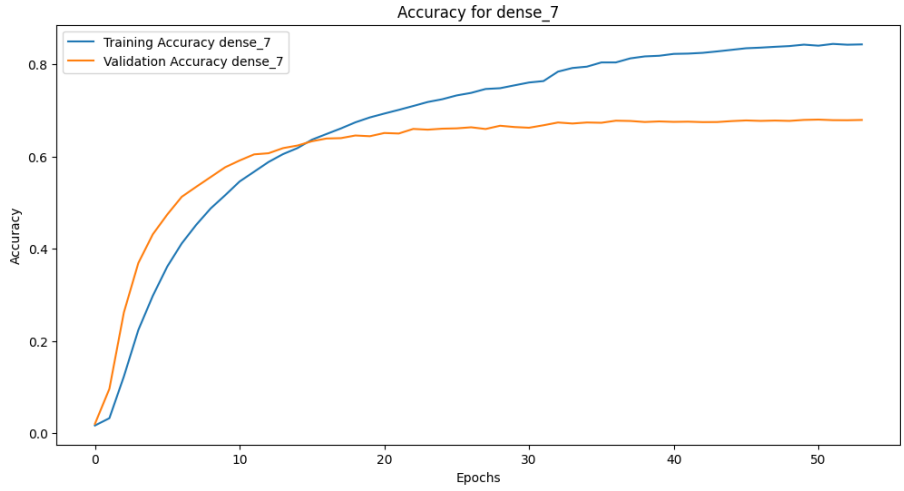

Figure 4.14: Training accuracy and validation accuracy of the fourth output layer

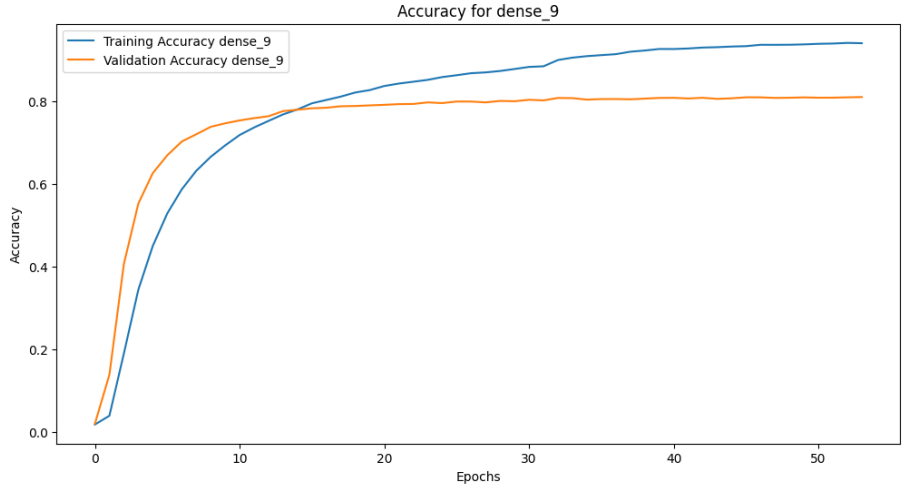

Figure 4.15: Training accuracy and validation accuracy of the fifth output layer

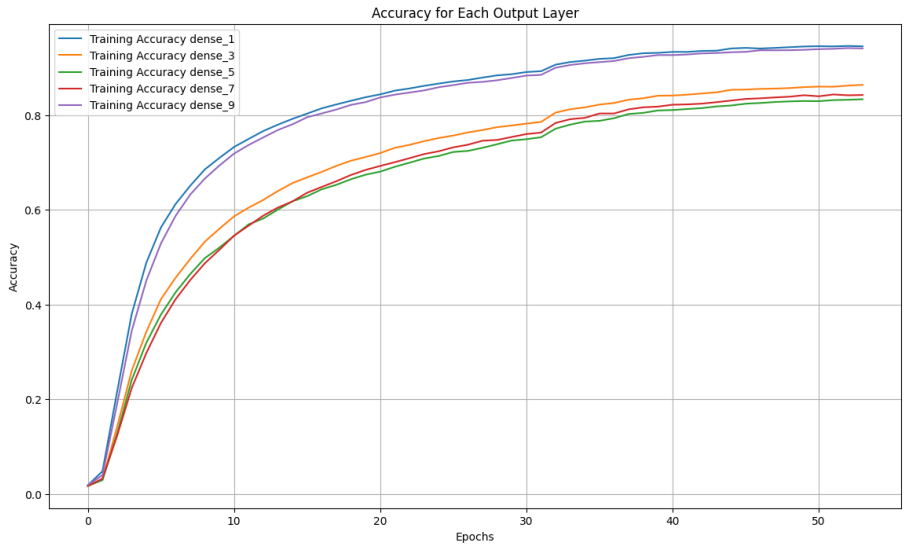

Figure 4.16: Accuracy of all the output layers

The plots show that for all the dense layers, the training accuracy improves as the epochs increase. The validation accuracy initially improves, and starts stagnating after around epoch 20.

Despite the increasing gap between the two metrics, a satisfying improvement has been achieved with respect to the first network architecture.

By plotting together the training loss and the validation loss of the model, we can clearly look at the difference in performance respect to the first architecture (Figure 4.17):

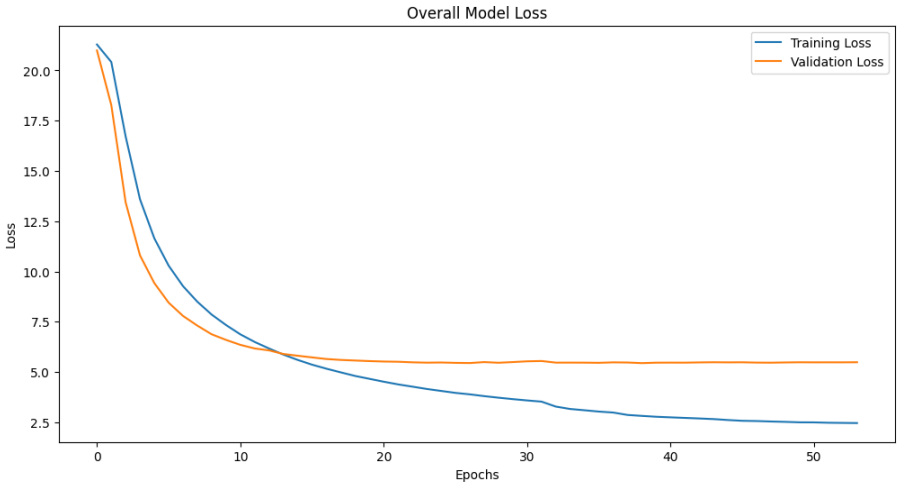

Figure 4.17: Overall model loss

Figure 4.16 shows the satisfactory levels of accuracy of the improved model.

Anyway, we can notice how the network struggles to perform well in the middle layers, as opposed instead to the outer ones (layers 1 and 5 against layers 2, 3 and 4). This is an interesting phenomenon, to which an explanation can be found in the structure of the content of the employed images.

By analyzing some samples it can be seen that it's not rare that some characters happen to be overlapped to some other characters to their right and left of the CAPTCHA. This can lead to more interference with which the network has to deal when trying to recognize the middle characters, as opposed instead to when dealing with the outer ones (Figure 4.18). Moreover, the pixels separating a character in the CAPTCHA from its surrounding ones are always very low in number.

Figure 4.18: Sample in which the third and fourth characters overlap. The label for this CAPTCHA is 'l4OMy'

4.2.6 Testing

The network has been tested on 10% of the whole dataset, consisting of 22613 CAPTCHA images.

As the plots in the previous paragraph show, the accuracy of each output layer, corresponding to how well the NN manages to recognize a character in a certain position of the CAPTCHA image (i.e. i-th layer will try to recognize the i-th CAPTCHA character), is reported in Table 4.5:

Character	Accuracy	Accuracy (Percentage)
1	0.8092	80.92%
3	0.6542	65.42%
4	0.6732	67.32%
5	0.8071	80.71%

Table 4.5: Accuracy for Different Characters

Given such values, the average accuracy of the output layers is 0.7260 or 72.60%.

By employing the proposed model to predict the samples of the test set, we can evaluate the prediction against the ground truth, and count, for example, the number of characters in the sample that the network was able to recognize.

# CAPTCHAs with **5** guessed characters	6185
# CAPTCHAs with **4** guessed characters	7297
# CAPTCHAs with **3** guessed characters	5106
# CAPTCHAs with **2** guessed characters	2818
# CAPTCHAs with **1** guessed characters	1022
# CAPTCHAs with **0** guessed characters	185

Table 4.6: Number of CAPTCHAs for which the NN was able to recognize a certain number of characters

Table 4.6 shows for how many CAPTCHAs the network was able to recognize all 5 characters, 4 characters, 3 characters, 2 characters and only 1 character. The data tells us that the network behaved in a rather satisfactory manner, being able to correctly recognize a number of characters greater or equal than 4 in 59.62% of the proposed CAPTCHAs.

Moreover, for 82.20% of the CAPTCHAs, most of the characters were correctly recognized (3, 4 and 5 characters).

The number of samples for which the network wasn't able to recognize any character is very low, stopping at just 185 CAPTCHAs.

Chapter 5

Conclusions

5.1 Results

We recall here the performance in terms of training loss against validation loss of the two networks:

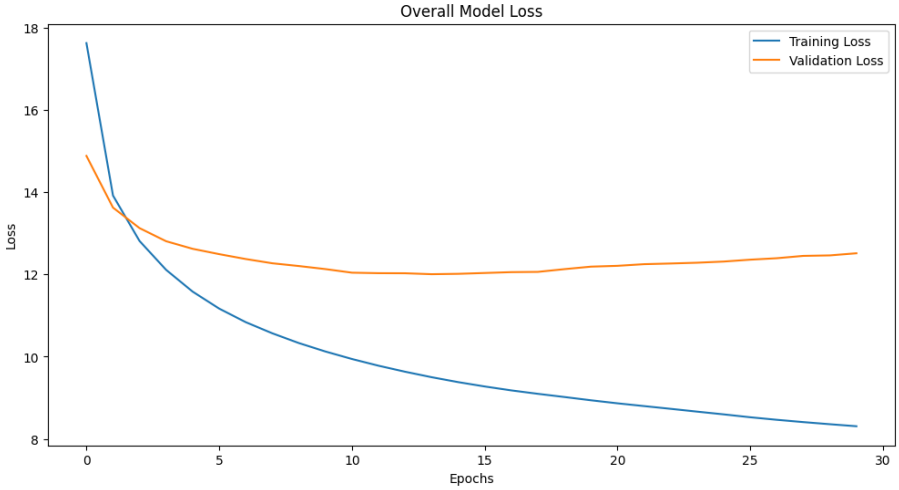

Figure 5.1: Overall model loss for the initial network

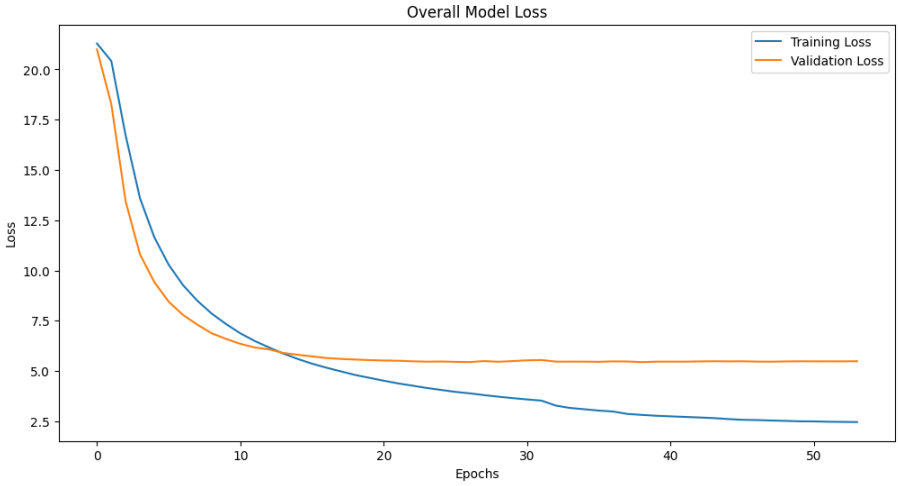

Figure 5.2: Overall model loss for the improved network

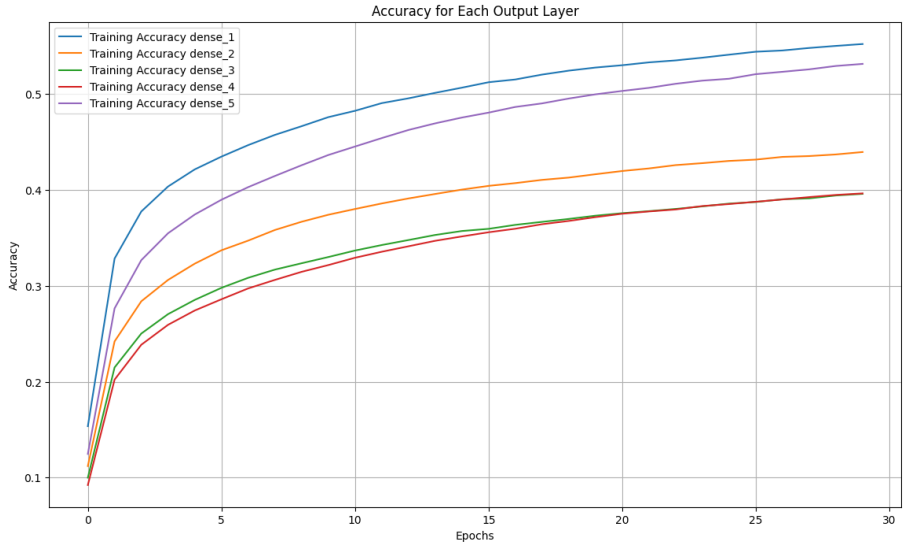

Figure 5.3: Accuracy of all the output layers in the initial network

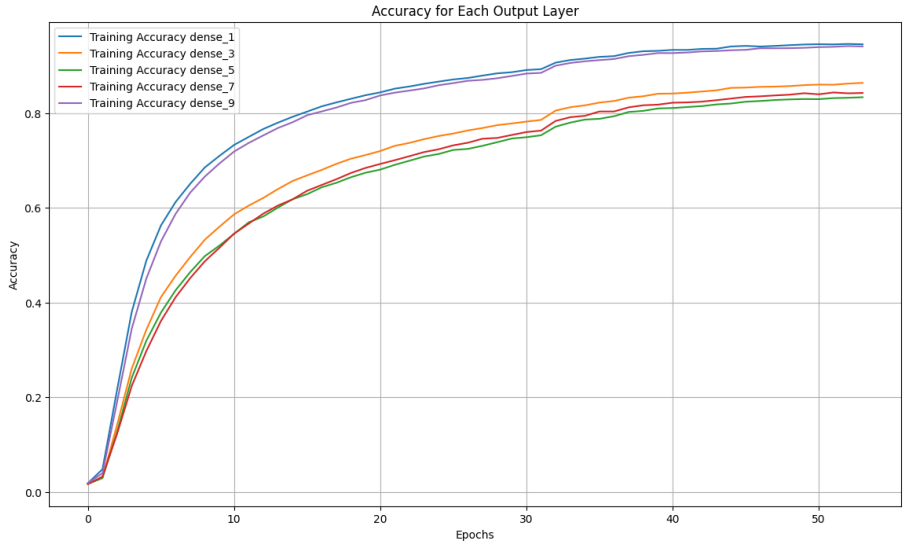

Figure 5.4: Accuracy of all the output layers in the improved network

The graphs show the improvement in performance that has been brought by the employment of regularization techniques in the second network. The choice of such techniques, along with a slight complication of the network architecture and the introduction of callbacks, has enabled us to reach satisfactory accuracy levels and reduce the overfitting phenomenon that was so high in the first proposed architecture.

5.2 Future Works

To further enhance the performance and robustness of the CAPTCHA recognition network, several areas of improvement can be explored:

- Employing advanced data augmentation techniques: while the current model leverages basic data augmentation strategies such as random rotation, zoom, and translation, more sophisticated techniques like elastic deformations, random erasing, and adversarial training could be incorporated to simulate more complex distortions. This may help the model generalize better to a wider variety of CAPTCHA challenges.
- Incorporating attention mechanisms: adding attention layers could allow the model to focus on the most relevant parts of the image for each character, improving its ability to handle CAPTCHAs with overlapping or noisy characters.
- Transfer learning: using transfer learning from pretrained networks on tasks like OCR (Optical Character Recognition) could provide a strong initial set of features, reducing the training time and improving accuracy.

Bibliography

[1] Ian Goodfellow, Yaroslav Bulatov, Julian Ibarz, Sacha Arnoud, and Vinay Shet. Multi-digit number recognition from street view imagery using deep convolutional neural networks. *arXiv preprint arXiv:1312.6082*, 2013.

[2] Max Jaderberg, Karen Simonyan, Andrew Zisserman, and Koray Kavukcuoglu. End-to-end text recognition with convolutional neural networks. *arXiv preprint arXiv:1412.2306*, 2014.

[3] Guohai Ye, Jianxin Li, and Xiang Liu. Captcha recognition using convolutional neural networks. In *2018 24th International Conference on Pattern Recognition (ICPR)*, pages 2428–2433. IEEE, 2018.

[4] Chao Zhang and Xi Xiao. Breaking text-based captchas using convolutional neural networks. *IEEE Transactions on Cognitive and Developmental Systems*, 2020.