Application Project Report

Digit Recogniser using Tensorflow backend

Project by: Ignatius Tang Hong-Quan

Supervising Professor: Professor Kai Warendorf

Esslingen University of Applied Sciences

01 January 2024

Project Topic: Computer Vision and Machine Learning

Hours Spent: 40 hours

Content Page

1. What is Tensorflow and why?

TensorFlow is an open-source machine learning library developed by the Google Brain team, designed to facilitate the development and training of machine learning models, particularly deep neural networks. In the context of computer vision, TensorFlow is widely used for tasks such as image classification, object detection, and image segmentation.

The benefits of using TensorFlow in computer vision projects are significant. Firstly, TensorFlow offers versatility, allowing developers to implement various neural network architectures for different computer vision applications. It provides efficient tools for neural network training, supporting GPU acceleration for faster model training, which is crucial when working with large datasets and complex models.

TensorFlow also facilitates transfer learning through pre-trained models and TensorFlow Hub, enabling developers to leverage knowledge gained from training on large datasets, saving time and computational resources. The library's scalability and deployment capabilities make it suitable for real-world applications, allowing models to be deployed on various platforms, including edge devices and cloud services.

Additionally, TensorFlow has a large and active community, extensive documentation, and supports multiple programming languages, making it accessible and well-supported for computer vision practitioners. Overall, TensorFlow's versatility, efficiency, and community support make it a powerful tool for developing and deploying computer vision models.

2. Environment Preparation

Jupyter Notebook

Jupyter Notebook is a versatile open-source web application highly esteemed for its interactive coding environment and unique blend of executable code and formatted text (Markdown). Supporting various programming languages, with Python being predominant, Jupyter enables seamless integration of rich visualizations and easy sharing of interactive documents. Its popularity is further reinforced by a vibrant community contributing extensions and plugins, making it a powerful tool for coding, documentation, and collaboration.

From my previous Data Science projects, I have already installed a python environment on Jupyter Notebook so I will be using it to develop this project.

3. Data Preparation

3.1 Importing Libraries

I imported several essential libraries for data processing, visualization, machine learning, and deep learning. Here's a brief explanation of the importance of each:

- **Pandas and NumPy**
    - Pandas (`pd`):** Essential for data manipulation and analysis, particularly for handling structured data in DataFrames.
    - NumPy (`np`):** Used for efficient numerical operations, providing support for arrays and matrices.

- **Data Visualization (Matplotlib, Seaborn)**
    - Matplotlib (`plt`), Matplotlib Image (`mpimg`), Seaborn (`sns`):** These libraries are crucial for creating visualizations, helping to explore and understand data through plots and graphs.

- **TensorFlow and Keras**
    - TensorFlow (`tf`): An open-source machine learning library, widely used for building and training deep learning models.
    - Keras (`keras`): A high-level neural networks API running on top of TensorFlow, simplifying the process of building and training models.

- **Text Processing (Tokenizer)**
    - Tokenizer: Part of the Keras library, it is used for converting text data into a format suitable for deep learning models.

- **Scikit-Learn**
    - Train-Test Split (`train_test_split`), Confusion Matrix (`confusion_matrix`):** From scikit-learn, these are essential tools for splitting datasets into training and testing sets and evaluating model performance.

- **Keras Utilities for Convolutional Neural Networks (CNN)**
    - To_categorical: Converts class vectors to binary class matrices, crucial for categorical classification tasks.
    - Sequential, Dense, Dropout, Flatten, Conv2D, MaxPool2D: Building blocks for constructing Convolutional Neural Networks (CNNs) using Keras.

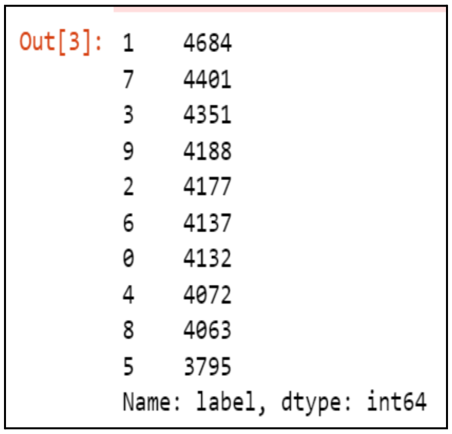
    - RMSprop (`RMSprop`): An optimizer often used for training neural networks.

- **Image Data Handling (ImageDataGenerator)**
    - ImageDataGenerator: Augments image data on-the-fly during model training, enhancing model generalisation.

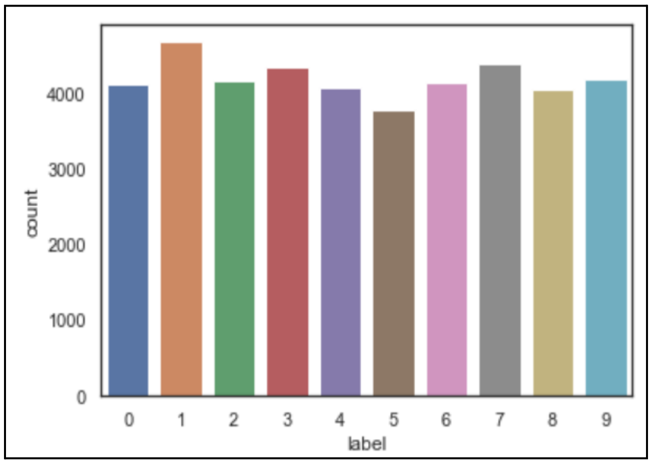
- **Learning Rate Reduction (ReduceLROnPlateau)**
    - ReduceLROnPlateau: A callback in Keras that adjusts the learning rate during training, improving model convergence.

3.2 Load data

Once we load both train & test .csv files, we check the counts for each digit from 0-9 using a countplot:



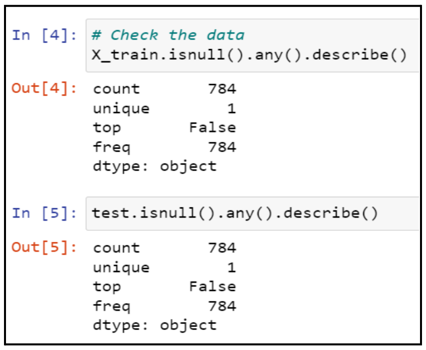
```
Out[3]: 1    4684
        7    4401
        3    4351
        9    4188
        2    4177
        6    4137
        0    4132
        4    4072
        8    4063
        5    3795
        Name: label, dtype: int64
```



^Figure 1: Countplot and count output

From the countplot above, there are approximately 4000 counts for all the digits.

3.3 Check for null and missing values

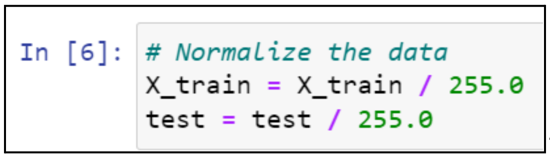


```
In [4]: # Check the data
        X_train.isnull().any().describe()

Out[4]: count       784
        unique        1
        top       False
        freq        784
        dtype: object

In [5]: test.isnull().any().describe()

Out[5]: count       784
        unique        1
        top       False
        freq        784
        dtype: object
```

BothBoth train and test data count = **784**.

No missing values, dataset is free of corrupted images, safe to proceed.

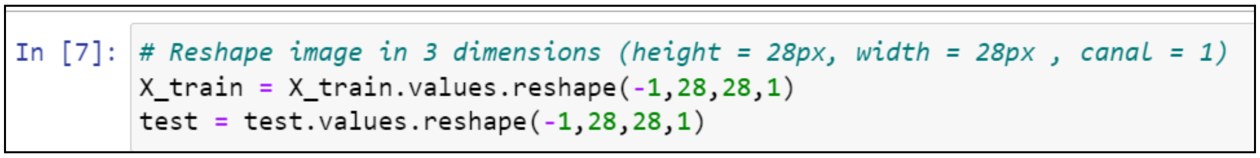
3.4 Normalisation



```
In [6]: # Normalize the data
        X_train = X_train / 255.0
        test = test / 255.0
```

Perform greyscale normalisation to reduce the effect of illumination's differences.
Moreover the CNN converges faster on [0..1] data than on [0..255].

3.5 Reshape

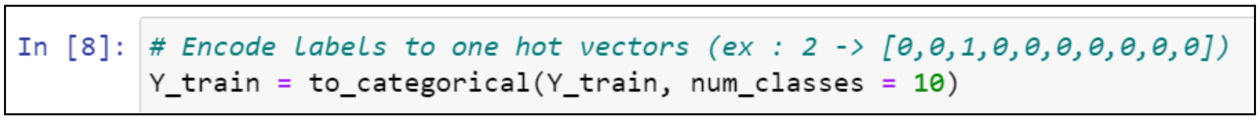


```
In [7]: # Reshape image in 3 dimensions (height = 28px, width = 28px , canal = 1)
        X_train = X_train.values.reshape(-1,28,28,1)
        test = test.values.reshape(-1,28,28,1)
```

`train` and `test` images (28px x 28px) have been stored into pandas.Dataframe as 1D vectors of 784 values. We reshape all data to 28x28x1 3D matrices.

Keras requires an extra dimension in the end which corresponds to channels. MNIST images are grey-scaled so it uses only one channel. For RGB images, there are 3 channels, we would have reshaped 784px vectors to 28x28x3 3D matrices.

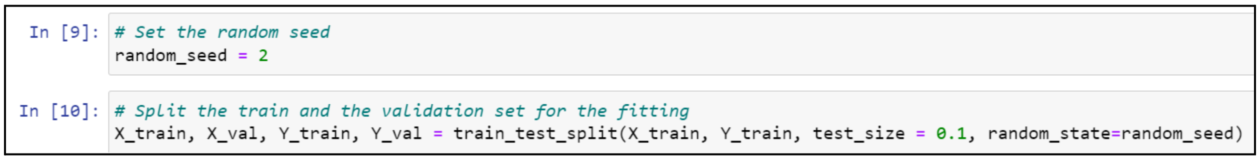
3.6 Label Encoding



```
In [8]: # Encode labels to one hot vectors (ex : 2 -> [0,0,1,0,0,0,0,0,0,0])
        Y_train = to_categorical(Y_train, num_classes = 10)
```

Labels are `10` digits numbers from `0` to `9`. We need to encode these labels to one hot vector (eg : 2 -> [0,0,1,0,0,0,0,0,0,0]).

3.7 Split Training and Validation Set

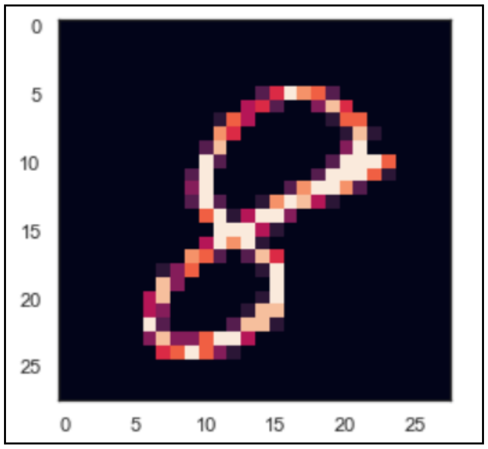


```
 In [9]: # Set the random seed
         random_seed = 2

In [10]: # Split the train and the validation set for the fitting
         X_train, X_val, Y_train, Y_val = train_test_split(X_train, Y_train, test_size = 0.1, random_state=random_seed)
```

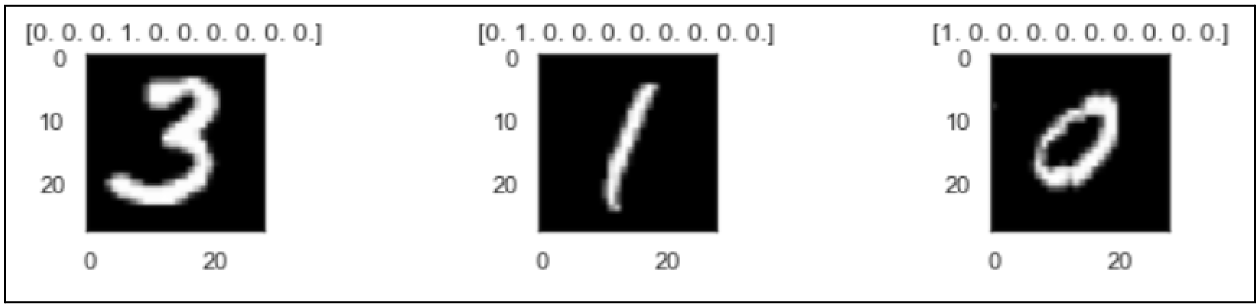
I chose to split the `train` set in two parts:

- a small fraction (10%) became the validation set which the model is evaluated and
- The rest (90%) is used to train the model.

Since we have 42,000 training images of balanced labels (see 2.1 Load data), a random split of the `train` set does not cause some labels to be over-represented in the validation set.



<Figure 2: example of train dataset



^Figure 3: example of raw train dataset

4. Train/Test Model

4.1 What is a Convoluted Neural Network(CNN) and how does it work?

A **Convolutional Neural Network (CNN)** is a type of deep neural network designed for processing structured grid data, such as images. CNNs are particularly effective in tasks like image recognition and classification. The key architectural components of a CNN include convolutional layers, pooling layers, and fully connected layers.

1. Convolutional Layers:
   - Convolutional layers apply filters (small matrices) to the input image, enabling the network to learn spatial hierarchies of features. These filters slide across the input image, performing element-wise multiplication and aggregation, capturing patterns like edges, textures, and shapes.

2. Pooling Layers:
- Pooling layers downsample the spatial dimensions of the input, reducing the computational load and enhancing translation invariance. Max pooling, for example, retains the maximum value within a local region, preserving the most significant features.

3. Fully Connected Layers:
- Fully connected layers process the high-level features extracted by previous layers and make final predictions. They connect every neuron to every neuron in the adjacent layers, enabling the network to learn complex relationships and patterns in the data.

4. Activation Functions:
- Activation functions, such as ReLU (Rectified Linear Unit), introduce non-linearities to the network, allowing it to model complex relationships and make non-linear predictions.

The overall workflow of a CNN involves passing an input image through multiple convolutional and pooling layers to extract hierarchical features. The extracted features are then flattened and passed through one or more fully connected layers for final classification or regression.

In the end, I used the features in two fully-connected (Dense) layers which is just an **Artificial Neural Networks(ANN)** classifier. In the last layer(Dense(10,activation="softmax")) the net outputs distribution of probability of each class.

My CNN architecture is:
In->[[Conv2D->relu]*2 -> MaxPool2D -> Dropout]*2 -> Flatten -> Dense -> Dropout -> Out

4.2 Set the Optimiser and Annealer

Once the layers are added to the model, I need to set up a score function, a loss function and an optimization algorithm.

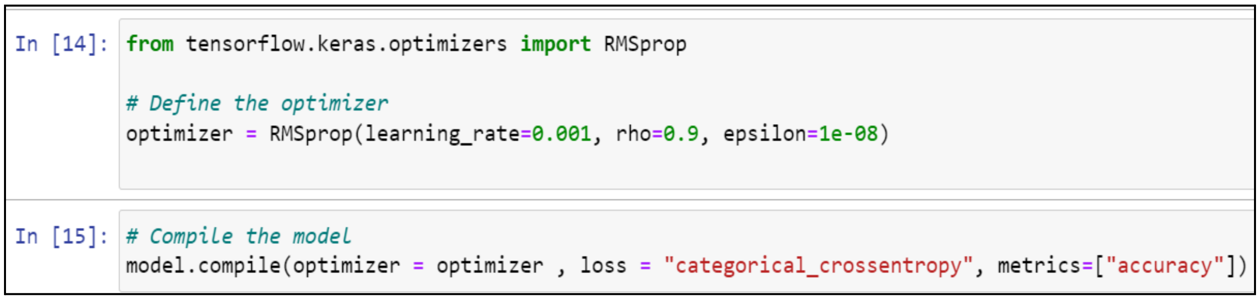
We define the loss function to measure how *poorly our model performs on images with known labels*. It is the error rate between the observed labels and the predicted ones.

We use a specific form for categorical classifications (>2 classes) called the "categorical_crossentropy".

The most important function is the optimizer. This function will iteratively improve parameters (filters kernel values, weights and bias of neurons ...) in order to minimise the loss.

> I chose **RMSprop** (with default values), it is a very effective optimizer. The RMSProp update adjusts the Adagrad method in a very simple way in an attempt to reduce its aggressive, monotonically decreasing learning rate. We could also have used the Stochastic Gradient Descent ('SGD') optimizer, but it is slower than RMSprop.

The metric function "*accuracy*" is used to evaluate the performance of our model. This metric function is similar to the loss function, except that the results from the metric evaluation are not used when training the model (only for evaluation).



```
In [14]: from tensorflow.keras.optimizers import RMSprop

         # Define the optimizer
         optimizer = RMSprop(learning_rate=0.001, rho=0.9, epsilon=1e-08)
```

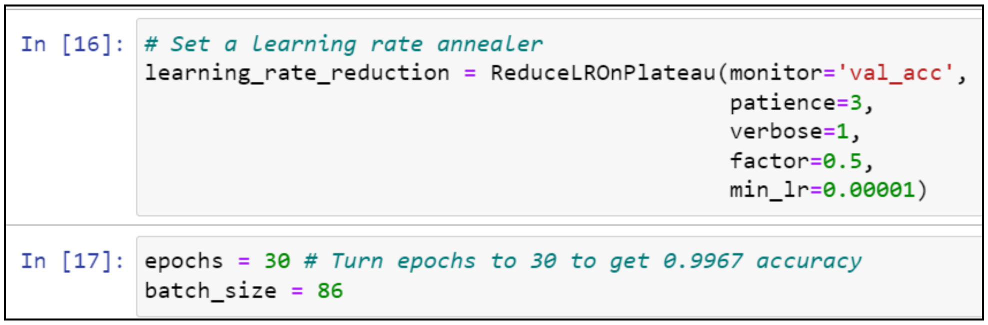
```
In [15]: # Compile the model
         model.compile(optimizer = optimizer , loss = "categorical_crossentropy", metrics=["accuracy"])
```

In order to make the optimizer converge faster and closest to the global minimum of the loss function, I used an **Annealing method of the learning rate (LR)**.

The LR is the step by which the optimizer walks through the '*loss landscape*'. The higher LR, the bigger are the steps and the quicker is the convergence. However the sampling is very poor with an high LR and the optimizer could probably fall into a local minima.

> Its better to have a decreasing learning rate during the training to reach efficiently the global minimum of the loss function.
>
> To keep the advantage of the fast computation time with a high LR, i decreased the LR dynamically every X steps (epochs) depending if it is necessary (when accuracy is not improved).

With the **ReduceLROnPlateau** function from Keras.callbacks, I chose to reduce the LR by half if the accuracy is not improved after 3 epochs.



```
In [16]: # Set a learning rate annealer
         learning_rate_reduction = ReduceLROnPlateau(monitor='val_acc',
                                                     patience=3,
                                                     verbose=1,
                                                     factor=0.5,
                                                     min_lr=0.00001)

In [17]: epochs = 30 # Turn epochs to 30 to get 0.9967 accuracy
         batch_size = 86
```

4.3 Data Augmentation

In order to avoid overfitting, I need to artificially expand the handwritten digit dataset. The concept involves applying minor transformations to the training data to replicate the variations that arise during the process of writing a digit.

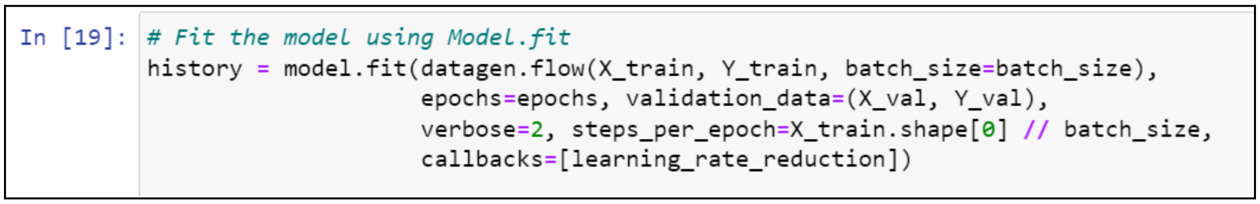
- For example, the number is not centred, the magnitude scale is not the same, the image is not oriented the same etc.

Augmentation:

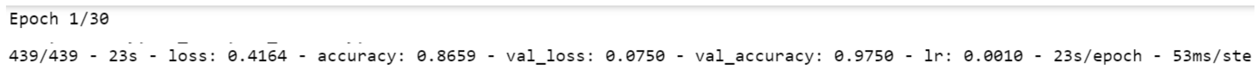
- Randomly rotate some training images by 10 degrees
- Randomly Zoom by 10% some training images
- Randomly shift images horizontally by 10% of the width
- Randomly shift images vertically by 10% of the height

I did not apply a vertical_flip nor horizontal_flip since it could have led to misclassification of symmetrical numbers such as 6 and 9.

Once the model is ready, fit the training dataset .

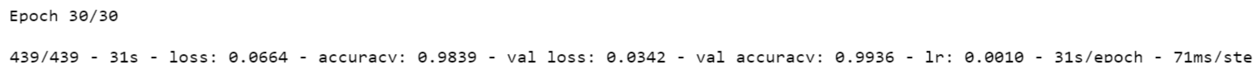


```
In [19]: # Fit the model using Model.fit
         history = model.fit(datagen.flow(X_train, Y_train, batch_size=batch_size),
                             epochs=epochs, validation_data=(X_val, Y_val),
                             verbose=2, steps_per_epoch=X_train.shape[0] // batch_size,
                             callbacks=[learning_rate_reduction])
```



```
Epoch 1/30
439/439 - 23s - loss: 0.4164 - accuracy: 0.8659 - val_loss: 0.0750 - val_accuracy: 0.9750 - lr: 0.0010 - 23s/epoch - 53ms/ste
```

1st epoch - accuracy = 0.8659

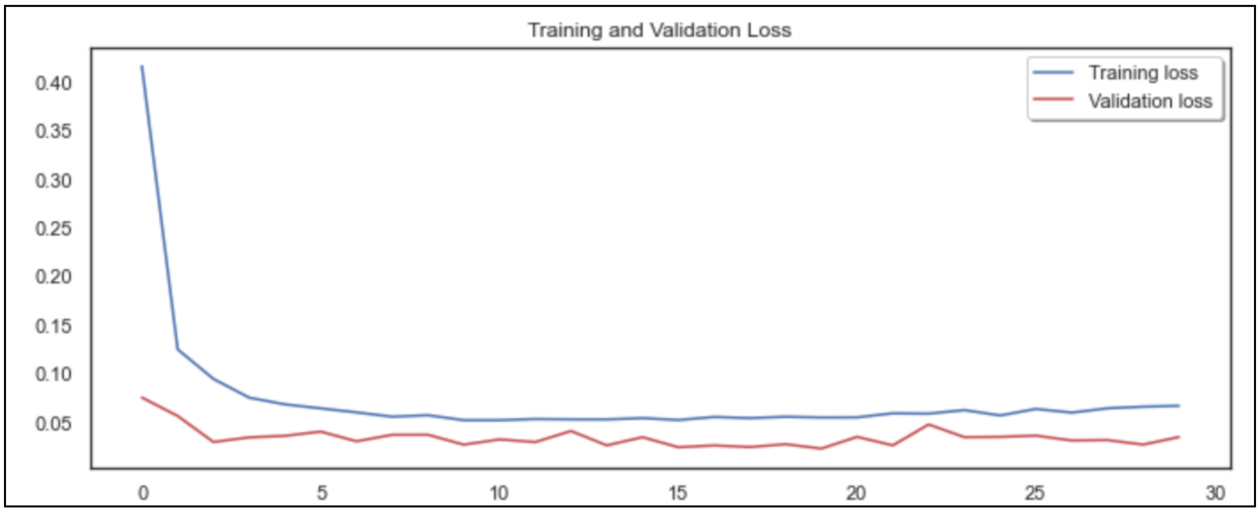


```
Epoch 30/30
439/439 - 31s - loss: 0.0664 - accuracy: 0.9839 - val_loss: 0.0342 - val_accuracy: 0.9936 - lr: 0.0010 - 31s/epoch - 71ms/ste
```

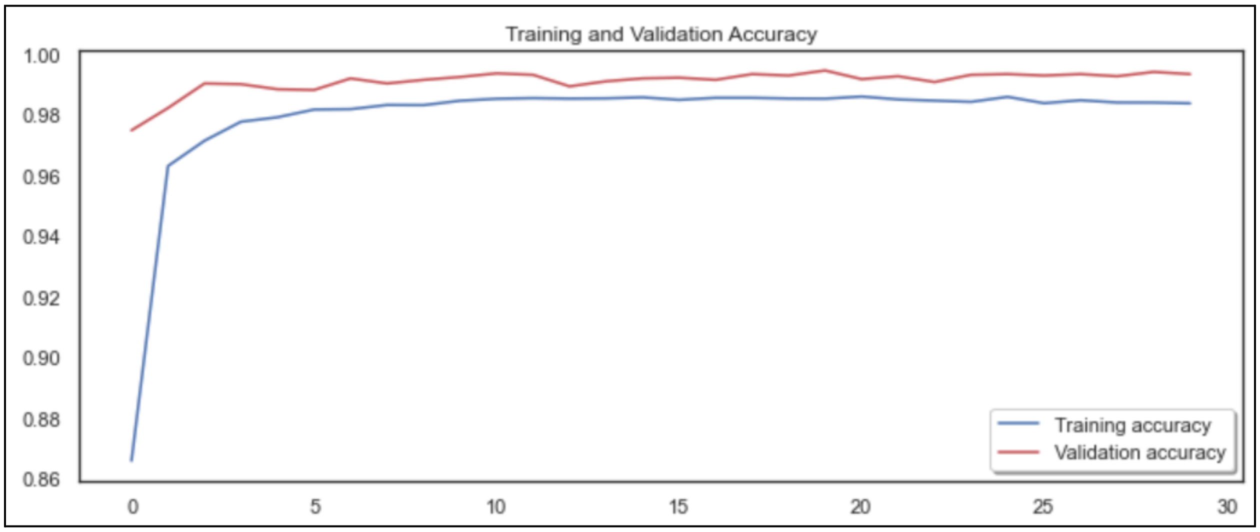
30th epoch - accuracy = **0.9839**

5. Model Evaluation

I then continue to plot the loss and accuracy curves for training and validation, followed by the confusion matrix. This allows us to better visualise the performance of the trained model.

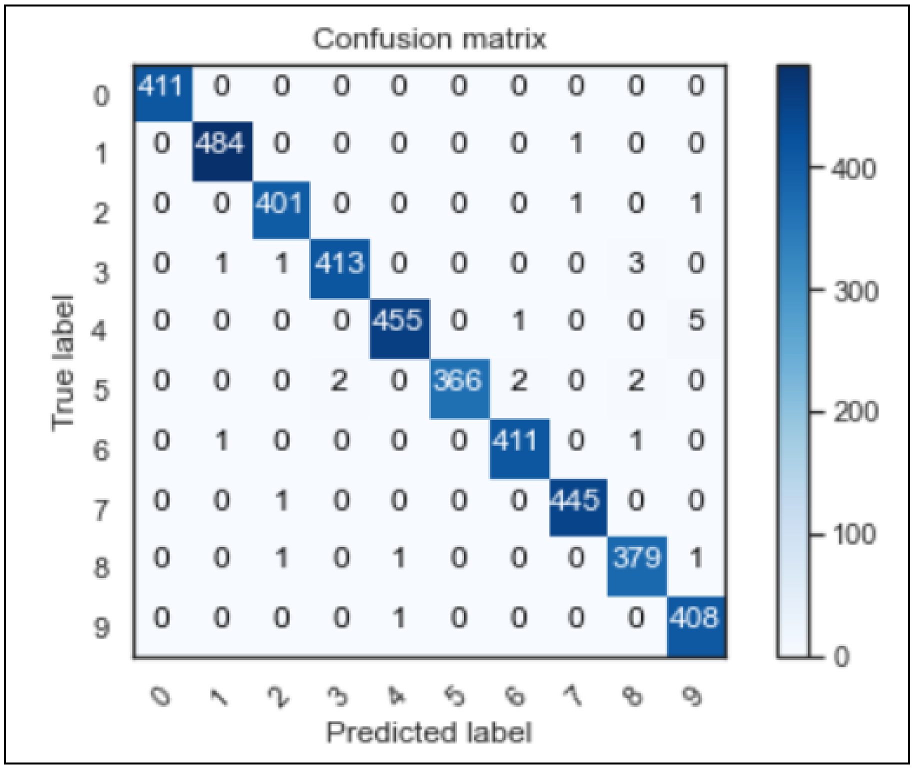


^Figure 4: Training and Validation Loss



^Figure 5: Training and Validation Accuracy

For both figure 4 & 5 above, training loss and accuracy generally plateaus after 10 epochs. Thereafter, increasing the number of epochs has minimal effect on the accuracy of the model prediction. Hence, this method brings diminishing returns.



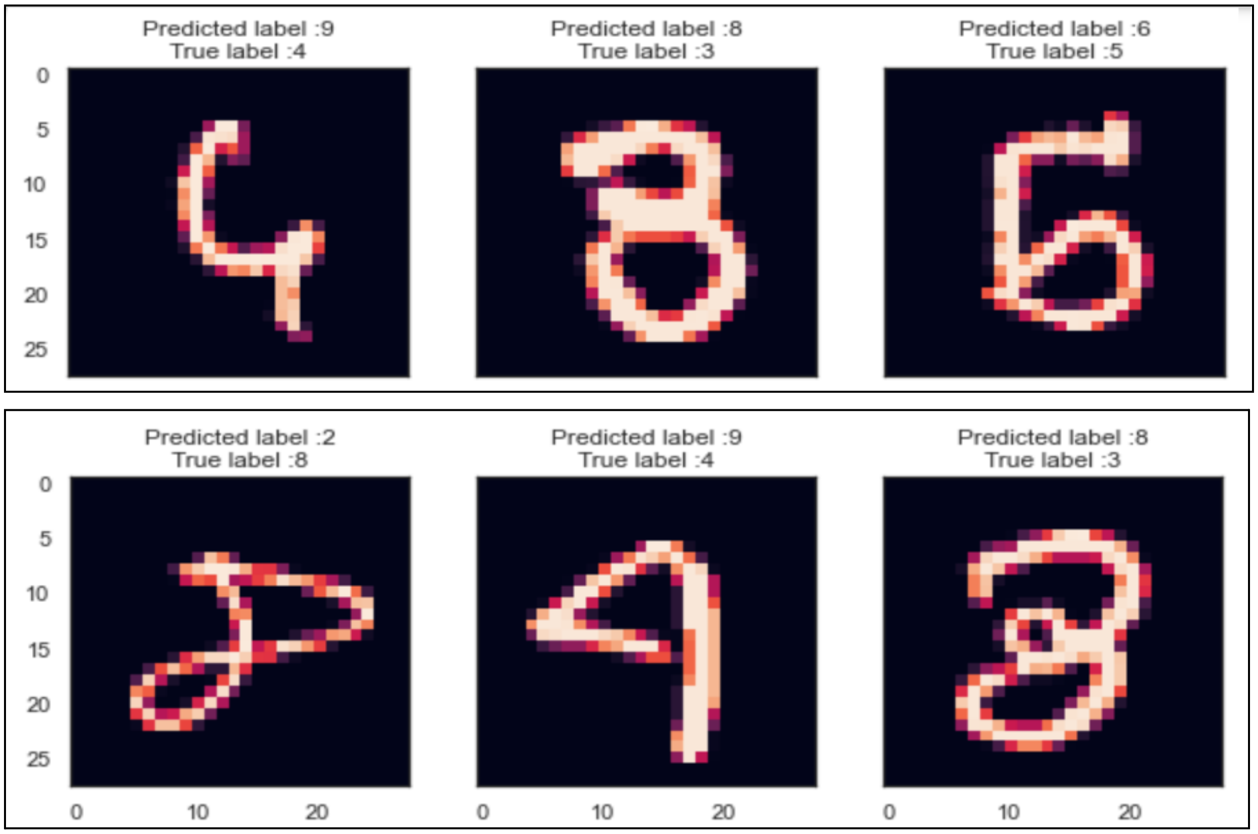
^Figure 6: Confusion Matrix of CNN model

Here we can see that our CNN performs very well on all digits with few errors considering the size of the validation set (4,200 images).

However, it seems that our CNN has some little troubles with the 4 digits, they are misclassified as 9. Sometimes it is very difficult to catch the difference between 4 and 9 when curves are smooth.

Let's investigate for errors.

I want to see the most important errors. For that purpose I need to get the difference between the probabilities of real value and the predicted ones in the results.The most important errors are also the most intricate.



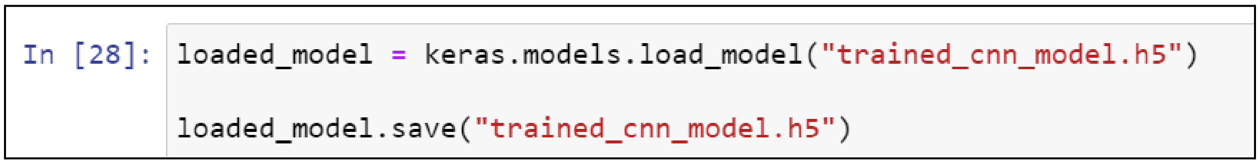
^Figure 7: Some Error Results

For these six cases, the model is not absurd. Some of these errors can also be made by humans. This only takes up a minute portion of the predictions, so the model is not as well-trained for poor handwriting.

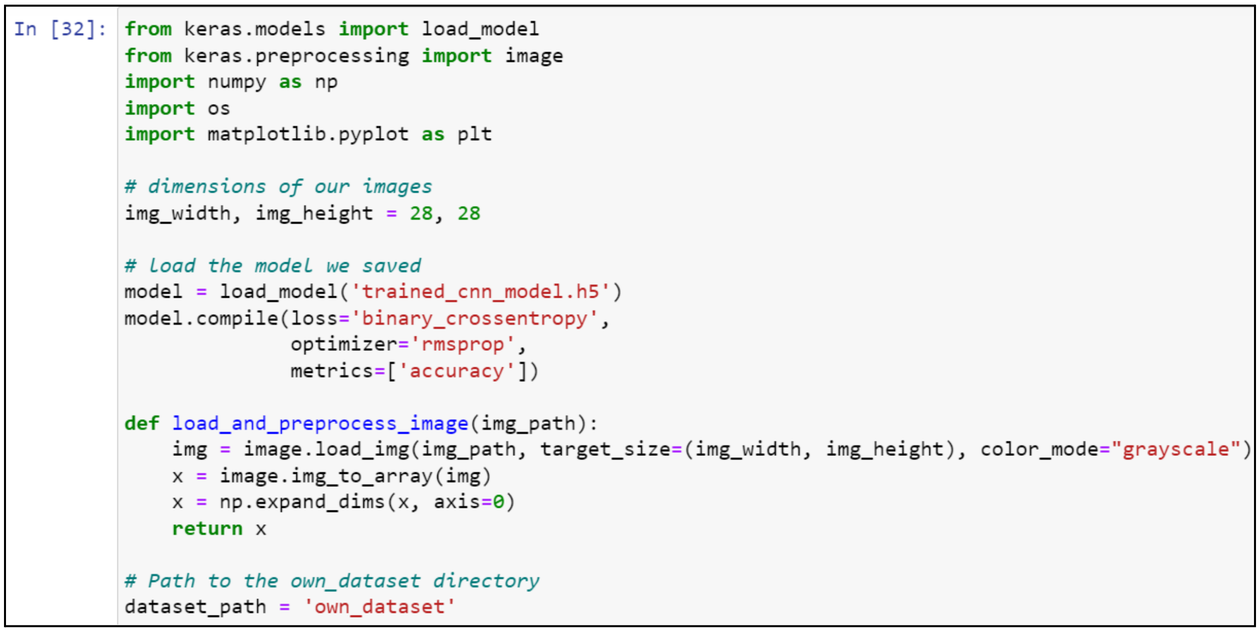
6. Real-Time Predictions

In order to test the reproducibility of the model, I compiled 3 unique sets of handwritten digits from 0-9 and ran the model against these newly loaded images. The images had to be manually **cropped**, **recentered** and **resized** to 28x28 pixels which I have done in pre-processing.*(images are saved under 'own_dataset' folder)*

Then I saved the trained model as a .h5 file for later usage.



```
In [28]: loaded_model = keras.models.load_model("trained_cnn_model.h5")

         loaded_model.save("trained_cnn_model.h5")
```

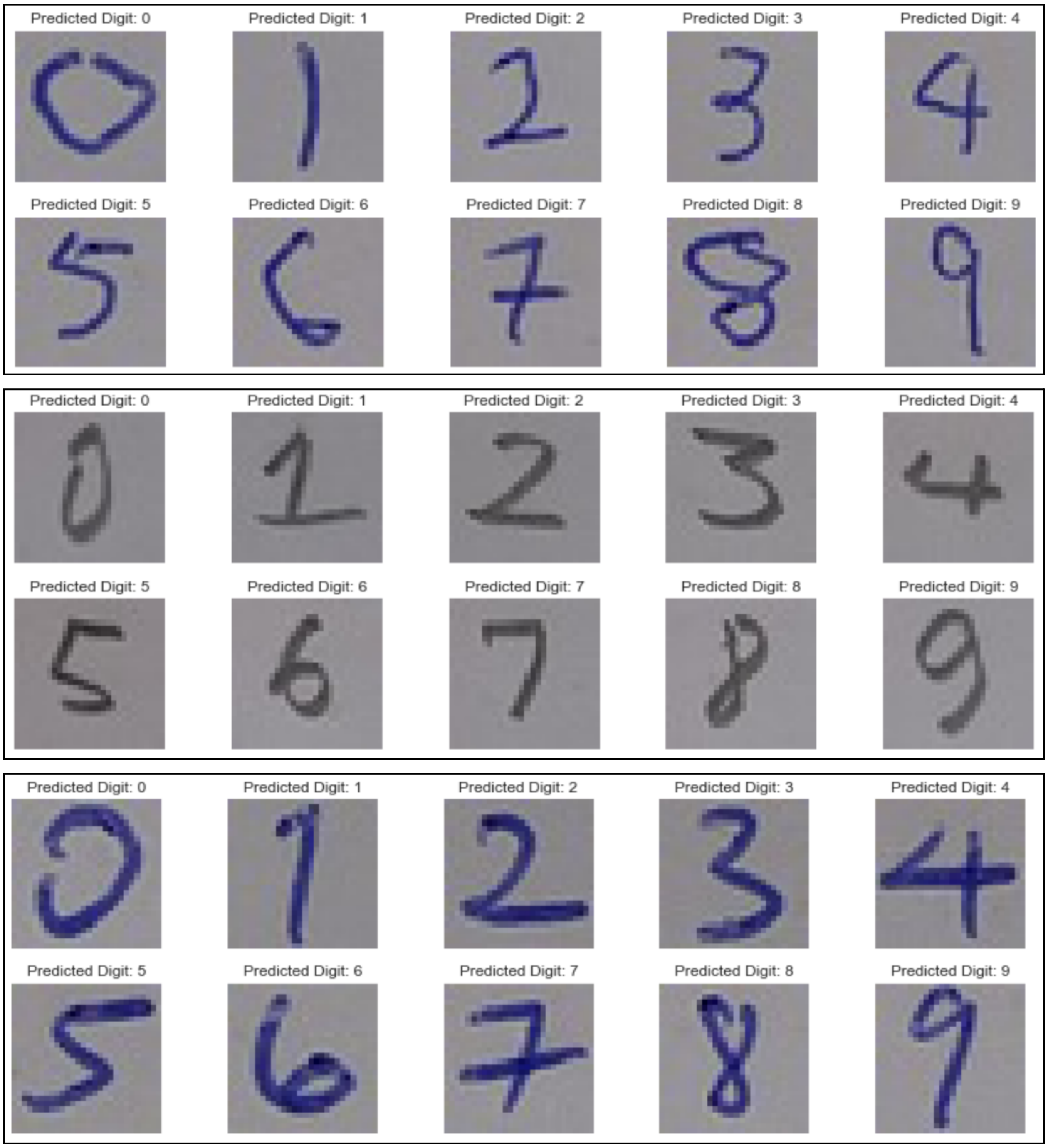


```
In [32]: from keras.models import load_model
         from keras.preprocessing import image
         import numpy as np
         import os
         import matplotlib.pyplot as plt

         # dimensions of our images
         img_width, img_height = 28, 28

         # load the model we saved
         model = load_model('trained_cnn_model.h5')
         model.compile(loss='binary_crossentropy',
                       optimizer='rmsprop',
                       metrics=['accuracy'])

         def load_and_preprocess_image(img_path):
             img = image.load_img(img_path, target_size=(img_width, img_height), color_mode="grayscale")
             x = image.img_to_array(img)
             x = np.expand_dims(x, axis=0)
             return x

         # Path to the own_dataset directory
         dataset_path = 'own_dataset'
```

The *load_and_preprocess_image* function ensures that the input images are set according to the trained model's compatibility, which is a **28x28 pixel grayscale image**, so that it can be resized and flattened properly by the CNN model.

```python
# Get a list of all image files in the directory
image_files = [f for f in os.listdir(dataset_path) if f.endswith('.jpg')]

# Set up the subplot grid
num_rows = 6
num_cols = 5
fig, axes = plt.subplots(num_rows, num_cols, figsize=(12, 12))

# Predict for each image in the dataset
for i, image_file in enumerate(image_files):
    img_path = os.path.join(dataset_path, image_file)
    x = load_and_preprocess_image(img_path)
    predictions = model.predict(x, batch_size=1)
    predicted_class = np.argmax(predictions, axis=1)[0]

    # Display the image in the subplot
    ax = axes[i // num_cols, i % num_cols]
    img = image.load_img(img_path, target_size=(img_width, img_height))
    ax.imshow(img, cmap='gray')
    ax.set_title(f"Predicted Digit: {predicted_class}")
    ax.axis('off')

# Adjust layout to prevent clipping
plt.tight_layout()
plt.show()
```

For loop iterates through each image in the *'own_dataset'* folder and runs the prediction for each digit in the image as shown on the following page.



^Figure 8: Output predictions of new dataset

Surprisingly, the model gave 100% accuracy for all the 30 counts of handwritten digits. However, feeding the model anomaly images such as poor lighting/ contrast would affect the accuracy of the model prediction. Hence, we should always pre-process the images before prediction.

7. Project Findings:

- Accuracy of prediction increases as the number of epochs increases, but significance of each consecutive epoch training becomes diminishing (especially after *10 epochs*)

- Running each epoch takes a significant amount of computational effort and time. Running 30 epochs would require about 7 minutes on average, hence 10 epochs is optimal for model training.

- For the model to be applied to a new dataset, the dataset must first be re-centred, resized and grey-scaled to fit the compatibility of the CNN-trained model, which can be done both automatically or manually.

- Model is not well-trained in recognising unconventional writings and poor quality images. (e.g. ambiguous handwritings, poor lighting/contrast)

- Misclassification of certain digits by model can also be made by humans. A solution could be the use of **Recurrent Neural Networks(RNNs)**, which is effective in capturing sequential dependencies in handwriting.

- Additional Data Augmentation techniques such as **Noise Reduction** can be further applied to improve performance of models.

8. Implications:

Similar training methods can be leveraged to train other Convolutional Neural Network (CNN) models for accurate image classification tasks. This concept is particularly applicable to alphabet recognition or character recognition in various languages. By extending the training methodology employed for digit recognition, models can be developed to accurately identify and classify characters, enabling a wide range of applications.

For instance, the current digit recognition model can serve as a foundation for building more sophisticated models that identify not only numbers but also letters or characters from different alphabets. This has significant implications for natural language processing tasks, optical character recognition (OCR), and document analysis. The model's ability to understand and interpret characters opens the door to extracting meaningful information from text-based content in images.

Moreover, the current digit recognition model can be further expanded to recognize not just individual characters but entire words or sentences. This is especially relevant in scenarios where handwritten or printed text appears in complex images, such as hand-written physical forms or documents. The extended model could accurately transcribe handwritten or printed text, making it valuable for applications in data entry, document digitization, and archival systems.

In essence, the training methods and principles applied in digit recognition can serve as a blueprint for developing versatile and adaptable models in the realm of computer vision. As the capabilities of these models evolve, they have the potential to play a crucial role in automating tasks related to image interpretation, language understanding, and information extraction from diverse visual data sources.

9. Conclusion:

In conclusion, the digit recogniser project revealed key insights into optimising the model's training process and understanding its limitations. The findings indicate that while accuracy improves with an increasing number of epochs, the marginal gain diminishes, and a practical trade-off is achieved at around 10 epochs, optimising computational efficiency. The preprocessing steps of re-centering, resizing, and grayscale conversion are crucial for ensuring the compatibility of the CNN-trained model with new datasets, offering flexibility for automated or manual implementation.

However, challenges persist in the model's ability to handle unconventional writings and poor-quality images, such as ambiguous handwriting or images with poor lighting and contrast. To address this, the project suggests exploring the use of Recurrent Neural Networks (RNNs) for capturing sequential dependencies in handwriting, potentially improving recognition performance. Additionally, incorporating advanced data augmentation techniques, including noise reduction, could further enhance the model's robustness.

Overall, the digit recogniser project provides valuable insights into model optimisation, dataset preprocessing, and potential avenues for future enhancements. Recognising the importance of balancing computational efficiency with model accuracy, the findings offer practical guidance for deploying the model effectively while acknowledging its limitations in handling challenging real-world scenarios.

10. References

Kaggle Link:https://www.kaggle.com/competitions/digit-recognizer/overview

1. *Tensorflow Core: Machine Learning for beginners and experts*. TensorFlow. (26.12.2023). https://www.tensorflow.org/overview

2. *Multilayer perceptrons for digit recognition with core apis : Tensorflow Core*. TensorFlow. (26.12.2023). https://www.tensorflow.org/guide/core/mlp_core

3. *Learn Computer Vision Tutorials*. Kaggle. (26.12.2023). https://www.kaggle.com/learn/computer-vision

4. *Human verification*. Stack Overflow. (26.12.2023). https://stackoverflow.com/search?q=digit%2Brecogniser%2Bwith%2Bkeras&s=49ef7921-2f24-48e7-9e70-ffc9a8d176d6

5. GeeksforGeeks. (2021, September 22). *Python: Classify handwritten digits with Tensorflow*. GeeksforGeeks. https://www.geeksforgeeks.org/python-classifying-handwritten-digits-with-tensorflow/