Utku Özbek

ICAT 3120 Technical Report : Digit Recognition Using Convolutional Neural Networks

Vaasa 2022

Contents

1 Introduction

Neural networks are machine learning algorithms that are at the very core of deep learning (IBM Cloud Education, 2020). A convolutional neural network is a type of neural network that is especially useful when the goal is to extract the features of an image in order to classify it.

This paper explains how a convolutional neural network can be used to build a model which can then be trained to recognize digit patterns in images. It examines how artificial neural networks can be utilized in training computers to recognize patterns in images from a computer science perspective. To this end, the case of building a neural network based, digit recognition application is studied using concepts from computer science and mathematics (mostly linear algebra and probability theory). The specifics of how this application is built is discussed in the second chapter “Building The Algorithm”, following this introduction. Subjects that are discussed here are the used tools, data preparation, data augmentation and constructing the model. The following third chapter provides details on the training process and model performance evaluation.

2 Building The Algorithm

This chapter will explain, in detail, how the algorithm is constructed. What tools are used and how the model is built is explained in this chapter of the paper.

2.1 Tools Used

Machine learning algorithms depend on significant amounts of data to function efficiently. In order to train a model capable of classifying digits, the MNIST dataset is used. This dataset, which is a subset of the larger NIST dataset, contains 60000 normalized, fixed-size images of handwritten digits (LeCun & Cortes, 2021). An additional 10000 images are provided with this dataset, that can be used for testing purposes.

The algorithm itself is written in Python as the libraries written in Python provide powerful deep learning tools. This algorithm uses Keras (working with TensorFlow background), Scikit-learn, Seaborn and Pillow libraries for the purposes of model building and model evaluation as well as data preparation.

2.2 Loading The Dataset

The MNIST dataset is available inside the Keras library. Importing the MNIST subsection of the library to allow easy access to the dataset. Accessing the dataset in this way eliminates the need of using additional Python libraries and manually splitting the training and testing subsets. The Python function that loads the dataset onto the training and testing arrays can be seen in **Picture 1**.

After the data is loaded as ndarrays (Numpy, 2022), the matrices are reshaped to the original image sizes. An additional fourth layer is added to the images here so that the model accepts these images as inputs. Following the adjustments, the arrays specifying

the labels for the testing and validation sets are made categorical (Keras, 2022). This is done in order to later compile the model with the categorical crossentropy loss function.

```
# Load dataset
def load_dataset():
    (x_train, y_train), (x_test, y_test) = mnist.load_data()
    x_train = x_train.reshape(-1, img_rows, img_cols, 1)
    x_test = x_test.reshape(-1, img_rows, img_cols, 1)

    y_train = to_categorical(y_train, num_classes)
    y_test = to_categorical(y_test, num_classes)

    return x_train, x_test, y_train, y_test
```

Picture 1. Function that returns that dataset in subsets (Brownlee, 2021).

2.3 Data Preparation

The data received from the MNIST dataset are examples of 28x28 pixels grayscale images of handwritten digits. The aim is to use convolution on the image matrices to reduce them to their features. Doing so will allow the neural network to learn from the image at hand.

The images have only one channel as they are grayscale. Thus, each image can be represented as a matrix with dimensions 28x28x1 (1x28x28x1 to be precise). Each cell (pixel) of these matrices hold a value ranging from 0 to 255. These values represent color. Normalizing these images can help the neural network to converge faster. Normalizing means compressing the value of the pixels to the [0,1] range as floats without losing information. This compression can be done as can be seen in **Picture 2**.

In the case of classifying an image that is not included in the dataset, some additional preparations must be made in order for the image to be accepted as input. The image must be resized to 28x28 pixels and it must be in grayscale. For accurate results, it is important to use images that have properties similar to the properties of the images in the MNIST dataset.

```
# Scale pixels
def prep_pixels(train, test):
    train_norm = train / 255.0
    test_norm = test / 255.0

    return train_norm, test_norm
```

Picture 2. Normalizing data (Brownlee, 2021).

2.4 Specifying Hyperparameters

Before constructing the model for the CNN, hyperparameters such as the batch size and number of epochs must be specified. The batch size variable is the number of samples the neural network will process before updating the weights (and/or biases) of the model (Brownlee, 2019). The number of epochs, on the other hand, represents the number of times the model goes through the training subset.

The values of these parameters are selected according to educated guesses. For this model, batch size is set to 64 and the number of epochs is set to 40 as can be seen in the Python code apparent in **Picture 3**.

```
# Specifying hyperparameters
batch_size = 64
num_classes = 10
epochs = 40
img_rows, img_cols = 28, 28
```

Picture 3. Specifying hyperparameters.

2.5 Data Augmentation

The data in the MNIST dataset were taken in similar conditions. Looking at the data, the differences between each image is small and as such there is little variety in the conditions in which the data was taken.

In a real world application, the aim is to create a model that is flexible. This means that the ideal model is not too sensitive on brightness, orientation, size or viewpoint. To account for such noise using only the data provided in the MNIST dataset is difficult. Therefore, to create a more flexible model, data augmentation must be utilized.

Data augmentation is the act of synthetically modifying the data at hand to diversify our data set (Gandhi, n.d.). Using data augmentation increases the amount of relevant data and reduces the chance of overfitting (i.e. memorization). The Python code for implementing data augmentation and the augmentation parameters for the digit recognition application can be seen in **Picture 4.**

```
# Data augmentation to prevent overfitting
datagen = ImageDataGenerator(
    featurewise_center=False,  # set input mean to 0 over the dataset
    samplewise_center=False,  # set each sample mean to 0
    featurewise_std_normalization=False,  # divide inputs by std of the dataset
    samplewise_std_normalization=False,  # divide each input by its std
    zca_whitening=False,  # apply ZCA whitening
    rotation_range=10,  # randomly rotate images in the range (degrees, 0 to 180)
    zoom_range = 0.1, # Randomly zoom image
    width_shift_range=0.1,  # randomly shift images horizontally (fraction of total width)
    height_shift_range=0.1,  # randomly shift images vertically (fraction of total height)
    horizontal_flip=False,  # randomly flip images
    vertical_flip=False)  # randomly flip images

train_gen = datagen.flow(x_train, y_train, batch_size=batch_size)
test_gen = datagen.flow(x_test, y_test, batch_size=batch_size)
```

Picture 4. Data augmentation (Kassem, n.d.).

Data augmentation for the MNIST dataset was done in the same way as in Kassem's (n.d.) model. The augmentation is done in this application by randomly rotating training images by 10 degrees, randomly zooming in on by 10% in some training images, randomly shifting images horizontally by 10% of the width and randomly shifting images vertically by 10% of the height. As Kassem (n.d.) points out, vertical and horizontal flips may result in misclassifications of the digits so these were not applied.

2.6 Constructing The Model

The construction of neural networks is done based on two criteria: simplicity and performance. The simplest model which provides the most accurate results is what is called the best model. The MNIST model was also built with this principle in mind.

```
# Creating and training the neural network
def create_and_train_model(x_train, x_test, y_train, y_test):
    # Create the model
    model = Sequential()

    input_shape = (img_rows, img_cols, 1)
    model.add(Conv2D(filters=64, kernel_size = (3,3), activation="relu", input_shape=input_shape))
    model.add(MaxPooling2D(pool_size=(2,2)))

    model.add(BatchNormalization())

    model.add(Conv2D(filters=128, kernel_size = (3,3), activation="relu"))
    model.add(MaxPooling2D(pool_size=(2,2)))

    model.add(BatchNormalization())

    model.add(Conv2D(filters=256, kernel_size = (3,3), activation="relu"))
    model.add(MaxPooling2D(pool_size=(2,2)))

    model.add(Flatten())
    model.add(BatchNormalization())

    model.add(Dense(512,activation="relu"))
    model.add(Dense(256,activation="relu"))
    model.add(Dense(num_classes,activation="softmax"))

    model.compile(loss=keras.losses.categorical_crossentropy,
                  optimizer="adadelta",metrics=['accuracy'])
```

Picture 5. Model overview (Kassem, n.d.).

The layers of the model can be seen in **Picture 5** above. The general build and the idea of utilizing three convolution layers was taken from Kassem's (n.d.) model but the total number of convolution layers and the number of final ”dense” layers are different here. The number of epochs has also been reduced to avoid overfitting.

In the first layer, 64 filters of 3x3 dimensions are applied to the images. For the activation function, ReLu is used. The number of filters applied is doubled in the following convolution layers in order to further downgrade the images to their features. In this way, the model learns only the essential features and the error caused by noise in the data is reduced.

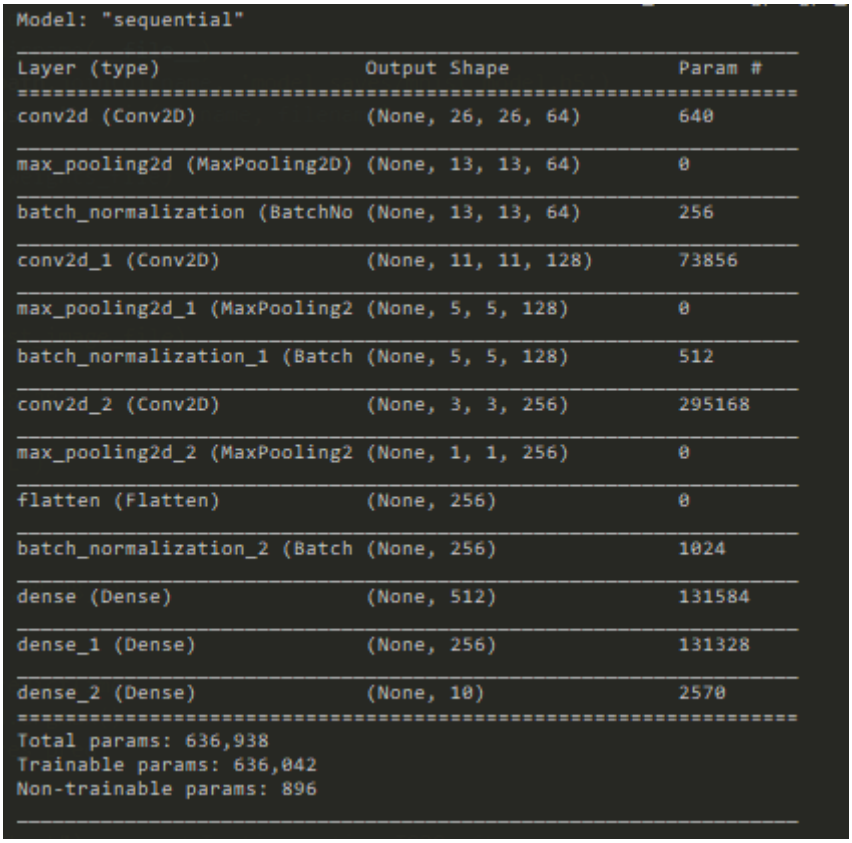
In between each convolution layer, the images go through max pooling and batch normalization. Max pooling is the calculation of the maximum value in each patch of each image matrix (Brownlee, 2019). This is done in order to extract the most relevant features in the feature map. Batch normalization, on the other hand, helps the model to learn better and faster as it normalizes the values in the feature maps. These processes are done together in order to stabilize the model and boost prediction accuracy.

After the final convolution layer, max pooling is applied one last time. At this point the images have been extremely simplified. When pooling is complete the matrices get ”flattened” into tensors in the layer called ”flatten” and they are once again normalized for the sake of faster convergence. These tensors are then plugged in as inputs for the following dense layers that house 256 and 512 neurons respectively. ReLu activation function is used once more in these layers. The dense layer following these two layers is the final layer of the model and it contains 10 neurons which then produce the results of the model by using the softmax function which turns the outputs of the previous layers into real numbers that sum to 1 as the objective is to calculate the probabilities of an input belonging to each class.

The loss function used for this model is categorical cross entropy. Cross entropy as a loss function is a way of calculating error in the model’s predictions (Brownlee, 2019). As there are more than 2 classes in the case of digit classification, categorical (i.e multi-class) cross entropy is used. By calculating the error in current predictions, the loss function informs the model of how the weights must be updated in order to achieve higher prediction accuracy.

3 Training The Model

The training of the model is done over 40 epochs. The process itself takes about 40 minutes with a Geforce RTX 2070 GPU. During this process, a total of 636,938 parameters are taken into consideration. Among these parameters, 896 of them are non-trainable meaning that these parameters are not optimized with this training. The information on the summary of the model as well as the types and numbers of parameters can be seen more clearly in the image below.

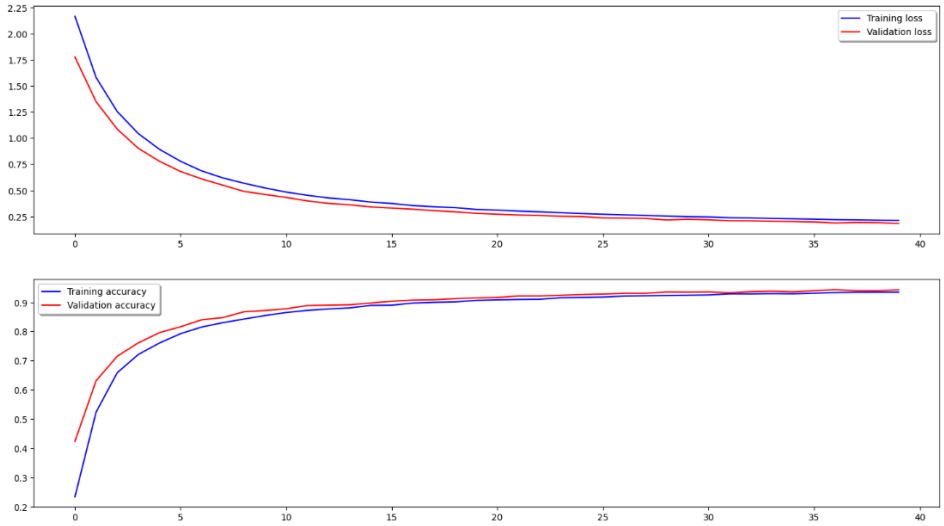


Picture 6. Summary of the model and parameters

3.1 Performance Evaluation

The algorithm for the digit recognition application is trained according to the model that was discussed in the second chapter. After 40 epochs of training, the model prediction accuracy increases from an initial accuracy of 41.5% to 95%. The increase in accuracy

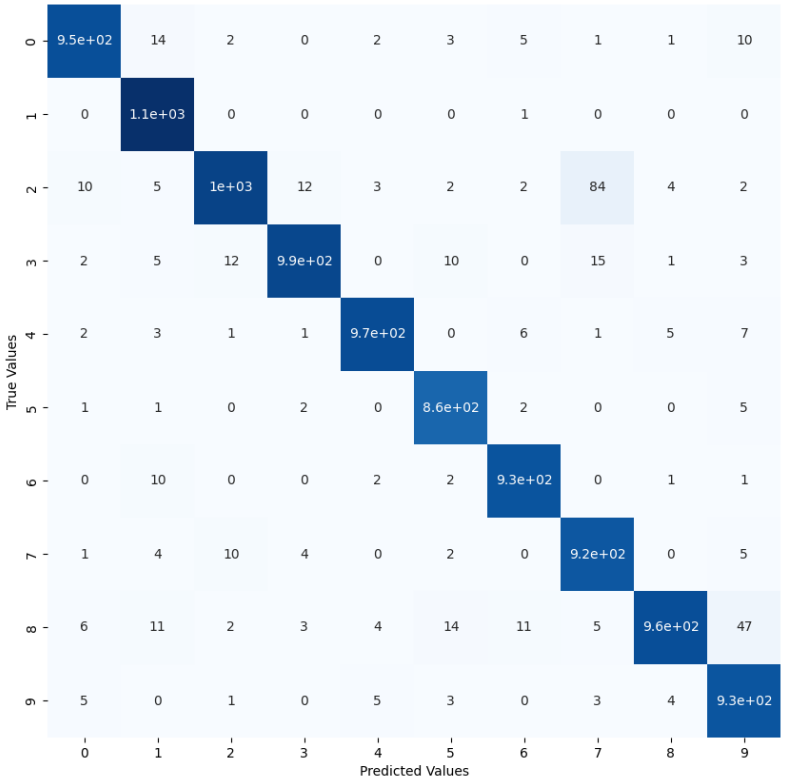
and the decrease in the output of the loss function (i.e. decrease in error) can be seen below in **Picture 7**.



Picture 7. Graphs detailing model performance over time.

The final accuracy can be increased to 99% by continuing the training for 50 epochs. However, doing this is risky since 99% accuracy may indicate a problem of overfitting. Adding more epochs may increase the apparent accuracy of the model while reducing flexibility. Causing overfitting is a problem in itself and a slight increase in accuracy at the cost of flexibility is also not desired. Hence, a model accuracy of 95% is optimal.

To evaluate this model's performance, plotting the confusion matrix of the validation results can also be useful. The confusion matrix can provide more insight into the error rate of the model and its recall and precision as well. This also means that the F-measure of the model can be derived from this matrix in order to quantify the model's performance rating. The confusion matrix of the validation results can be seen in the image below.



Picture 8. Confusion matrix of the validation results

3.2 Loading Images For Classification

At this point into the project, the application is ready to classify images taken in the real world since the model training is over. Before classifying these images, they are first downsampled to the size of 28x28 pixels. They are then grayscaled and another 4^{th} dimension is added to them so that the model can accept these images as inputs (final images are 1x28x28x1). After these steps, the values in the matrices are normalized and then finally the images become ready for classification.

4 Conclusion

Machine learning has many applications. One such application is pattern recognition. Through the use of machine learning algorithms, computer science knowledge and mathematical concepts it is possible for computers to differentiate between objects and concepts described in images.

For effective pattern recognizing models, a sub-field of machine learning that is called deep learning is used. With a deep learning method that is called a convolutional neural network (CNN) it is possible to reduce images to their simple features, making it easier for computer algorithms to derive the relation between the inputs and their corresponding outputs. After reducing the images to their features, an artificial neural network can be utilized to classify images.

In this paper, the details of implementation were discussed by creating an application that can classify digits from 0 to 9. The deep learning model built for this application makes use of CNNs in addition to more classical machine learning methods. In conclusion, it can be seen that CNNs can be utilized in image classification by extracting features from images in order to simplify them and help the weights of the model to converge faster by allowing algorithms to process more relevant data.

References

Brownlee, J. (2019, January 16). A Gentle Introduction to Batch Normalization for Deep Neural Networks. Machine Learning Mastery.

Retrieved 2022.03.01

https://machinelearningmastery.com/batch-normalization-for-training-of-deep-neural-networks/

Brownlee, J. (2019, October 21). A Gentle Introduction to Cross Entropy for Machine Learning. Machine Learning Mastery.

Retrieved 2022.03.01

https://machinelearningmastery.com/cross-entropy-for-machine-learning/

Brownlee, J. (2019, April 22). A Gentle Introduction to Pooling Layers for Convolutional Neural Networks. Machine Learning Mastery.

Retrieved 2022.03.01

https://machinelearningmastery.com/pooling-layers-for-convolutional-neural-networks/

Brownlee, J. (2019, October 26). Difference Between a Batch and an Epoch in a Neural Network: What Is the Difference Between Batch and Epoch. Machine Learning Mastery.

Retrieved 2022.02.26

https://machinelearningmastery.com/difference-between-a-batch-and-an-epoch/

Brownlee, J. (2021, November 14). How to Develop a Convolutional Neural Network From Scratch for MNIST Handwritten Digit Classification. Machine Learning Mastery.

Retrieved 2022.02.26

https://machinelearningmastery.com/how-to-develop-a-convolutional-neural-network-from-scratch-for-mnist-handwritten-digit-classification/

Gandhi, A. (n.d.). Data Augmentation: How to use Deep Learning when you have Limited Data – Part 2. Nanonets.
Retrieved 2022.02.27
https://nanonets.com/blog/data-augmentation-how-to-use-deep-learning-when-you-have-limited-data-part-2/

IBM Cloud Education. (2020, October 20). Convolutional Neural Networks. IBM.
Retrieved 2022.02.25
https://www.ibm.com/cloud/learn/convolutional-neural-networks

Kassem. (n.d.). MNIST: Simple CNN Keras (Accuracy : 0.99)=>Top 1%. Kaggle.
Retrieved 2022.02.27
https://www.kaggle.com/elcaiseri/mnist-simple-cnn-keras-accuracy-0-99-top-1/notebook

Keras. (2022, February 23). Python & Numpy utilities.
Retrieved 2022.02.26
https://keras.io/api/utils/python_utils/

LeCun, Y. & Cortes C. (2021, March 3). MNIST Database of handwritten digits. MNIST Database.
Retrieved 2022.02.25
http://yann.lecun.com/exdb/mnist/

Numpy. (2022, February 7). Numpy.ndarray.
Retrieved 2022.02.26
https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html