Mircea Maria-Madalina, 246, ICA

Virtual Reality handwritten character recognition using a Convolutional Neural Network

Problem statement: Recognize characters written in a virtual reality environment using a CNN.

VR project at: https://github.com/MadalinaMircea/VRCharacterRecognition

CNN project at: https://github.com/MadalinaMircea/MLCharacterRecognition

Flow description:

The virtual reality environment is developed in Unity using C#.

The CNN is developed in C#.

In VR, the used is presented with a whiteboard and a pen. They can pick up the pen and write a letter on the whiteboard. When the used has not written anything for 2 seconds, the whiteboard is cleared and the letter is sent to the CNN for recognition.

The Unity project and the CNN communicate through a folder called “UnityCNN” created on the computer desktop. Unity saves the texture from the whiteboard to the file “image.jpg”. The CNN continuously waits for such a file to exist. When it exists, the CNN reads it and analyses the image, deletes this file and creates a file called “response.txt” where the recognized character is written. The Unity project then reads this file and displays the response in a 3D text object in the VR environment.

The Unity VR project:

Developed for Oculus Rift S using SteamVR.

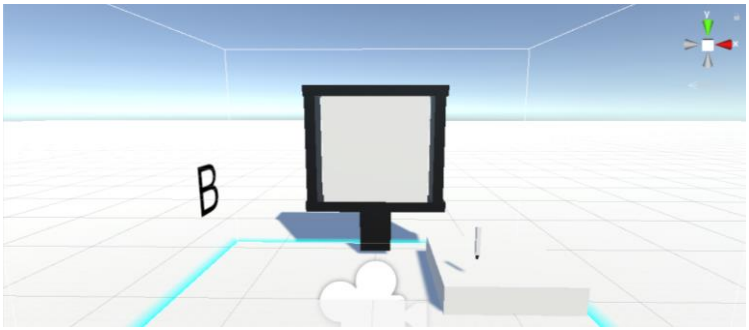
The “Hand” script is added to both controllers. This script makes the controllers able to grab an object.

The “Interactable” script is added to the pen. This allows the pen to be grabbed.

The “Whiteboard” script is attached to the whiteboard object. This controls the writing on the texture of the object by setting the pixels around a collision point to a certain color. This also counts the seconds from the last time the pen touched the board to see if it is time to send the texture to the CNN.

The “WhiteboardMarker” script creates a raycast from the tip of the pen upwards to check if the pen is close to an object tagged with the “Whiteboard” tag. If so, the pen calls methods in the “Whiteboard” script of this object and transmits the position of the collision.

The “CNNController” script controls when the texture image is sent to the CNN and when the response character is displayed.



Screen capture from the Unity VR project

The Convolutional neural network:

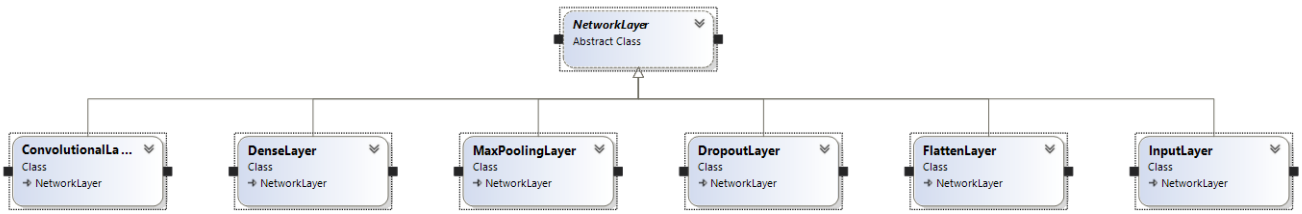
Developed from scratch (the Keras model is used for demonstrative purposes).

The CNNController file creates a ConvolutionalNeuralNetwork model and controls the training, testing and overall usage of this model.

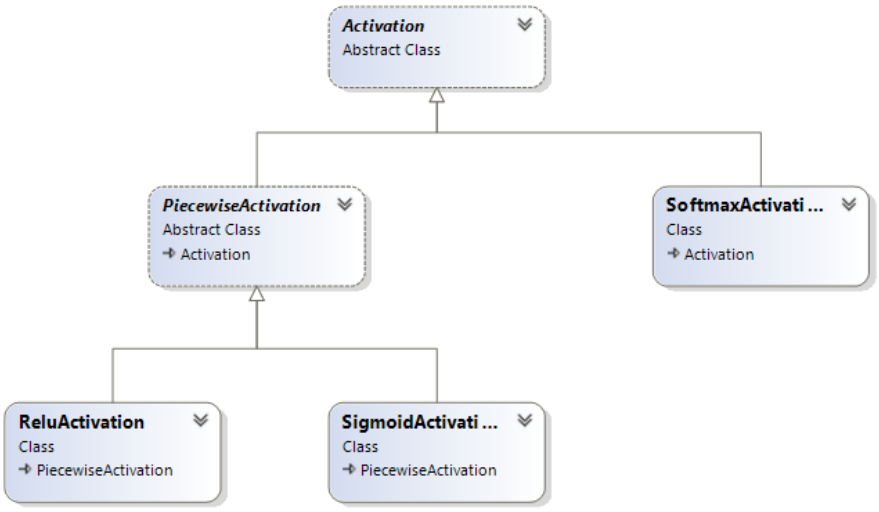
The model architecture is written to file as a json. The weights are written in a folder, with separate folders for each layer containing weights (convolutional and dense). Each layer folder contains separate files for each filter/unit. The filters and units are written to file as json. The same file-folder structure is used when reading a model.

The ImageProcessing folder contains the files needed for processing images. The ImageProcessing file contains methods that convert a JPG image to a matrix of numbers, normalize the matrix and so on. The LayerOutput class is a parent class for the FilteredImage and FlattenedImage classes. FilteredImage is an image with multiple channels, each channel being represented by a matrix. FlattenedImage is an image that has been flattened to one array. These classes are used as output in the layers of the network. The ImageRepository and ImageController classes are used to control the flow of the dataset (training, testing and validation) from the folders.

The CNN folder contains all the needed filed. The Layers folder contains the layer classes (Convolutional, Dense, Flatten, Dropout, Maxpooling), as well as the units needed in these layers (Filter, Kernel, Unit). All of the layers are children of the NetworkLayer class.



The Activations folder contains all of the activation files. The Activation class is the parent class. Softmax is a child of this class. PiecewiseActivation is also a child of the Activation class, but this is in itself a parent of all the other activations (Relu, Sigmoid). This parent class is needed for the activations that are applied element by element on a layer, not on the whole layer, like Softmax. The parsing of the elements is done in this parent class, while the element activation is overridden in the children classes.



The Utils folder contains helper classes (JsonHelper), as well as classes with mathematical computations used throughout the algorithm (Convolution, Matrix rotation).

Creating a model:

Before creating a model, the GlobalRandom.InitializeRandom() static method needs to be called. This is a static class used to generate all of the random numbers needed in a model. A global random was necessary because this static class will only request one seed for the random number generator, thus insuring that the kernels/weights will not be equal to each other (if each kernel has its own Random variable, there is a chance the seed will be equal among kernels and the kernels will end up equal to each other).

To create a model, a ConvolutionalNeuralNetwork object is created. Then, the “Add” method can be used to add any type of layer to it, since this method receives a “NetworkLayer” parameter, which is a parent of all the layer classes.

Before using the model, the Compile method needs to be called. This calls the Compile method of each layer with the previous layer as parameter, creating a link between layers and ensuring that the kernel/unit number is correct for all layers (the kernel number of a convolution layer should be equal to the filter number of the previous layer; the output matrix size of a

maxpooling layer will be equal to the size of the previous layer matrix divided by the pool size; etc).

Training a model:

To train a model, a list of InputOutputPair is given to the Train method. The InputOutputPair contains the path of a file, the character this image file represents, as well as the one-hot-encoded version of this character. This list has already been shuffled. The Train method parses the list, takes the image file and calls ImageProcessing to request a normalized FilteredImage version, the feeds this image to the network. The output is taken, the error between the output and the expected output is computed, then this error is passed to the Backpropagate method of each layer, starting with the last one.

Datasets used:

- Chars74K
- NIST
- original characters generated in the Windows Forms application

Dataset split:

- Training: 40 * Chars74K/letter + 1160 * NIST / letter (+ 30 * original / letter) => 1230 images/letter => 31980 images
- Testing: 10 * Chars74K/letter + 150 * NIST / letter (+ 10 * original / letter) => 170 images/letter => 4420 images
- Validation: 5 * Chars74K/letter + 190 * NIST / letter (+ 10 * original / letter) => 205 images/letter => 5330 images

Convolution operation:

For the zero-padded version, the matrix was not padded, but if statements were used to ensure that the indices were inside the bounds. This saves the computation time that would have been required by the padding.

The outer for-loops parse the output matrix. The inner for-loops parse the kernel. For the version with “same” padding, the following multiplications are summed:

```
matrix[i + kernelI - halfSize, j + kernelJ - halfSize] * kernel[kernelI, kernelJ];
```

For the version with “valid” padding, the following multiplications are summed:

```
matrix[i + kernelI, j + kernelJ] * kernel[kernelI, kernelJ];
```

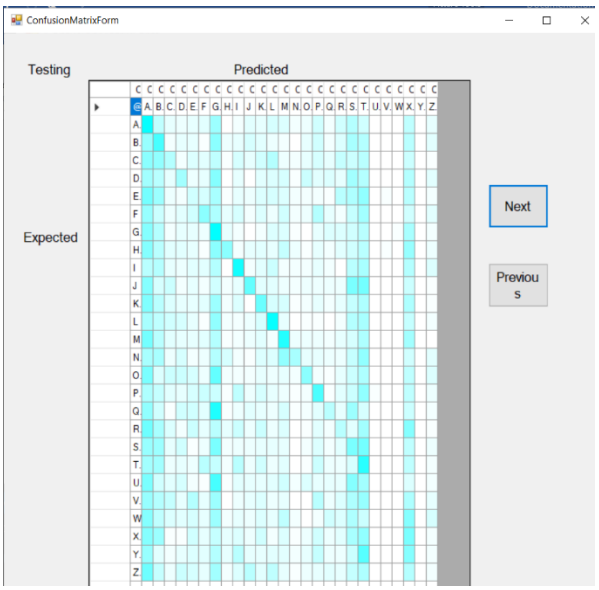
Performance:

Training for one image takes around 4 seconds.

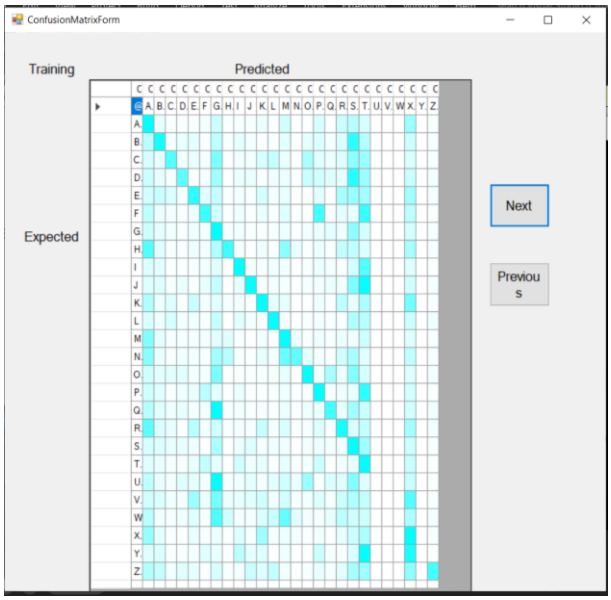
Recognising one image takes around 1 second.

These statistics were obtained after introducing parallelism in the computations. The C# Task class was used to parallelize computations. Each filter is computed in parallel, and each kernel from one filter is also computed in parallel. Also, for all of the 4-piece for-loops, the first for loop created tasks for each of the steps, then these tasks were waited for outside of the for loops.

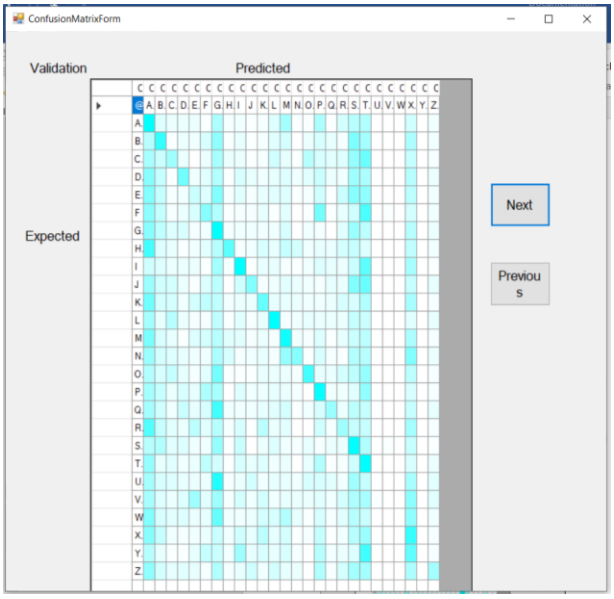
Confusion matrix



Confusion matrix for the testing set.

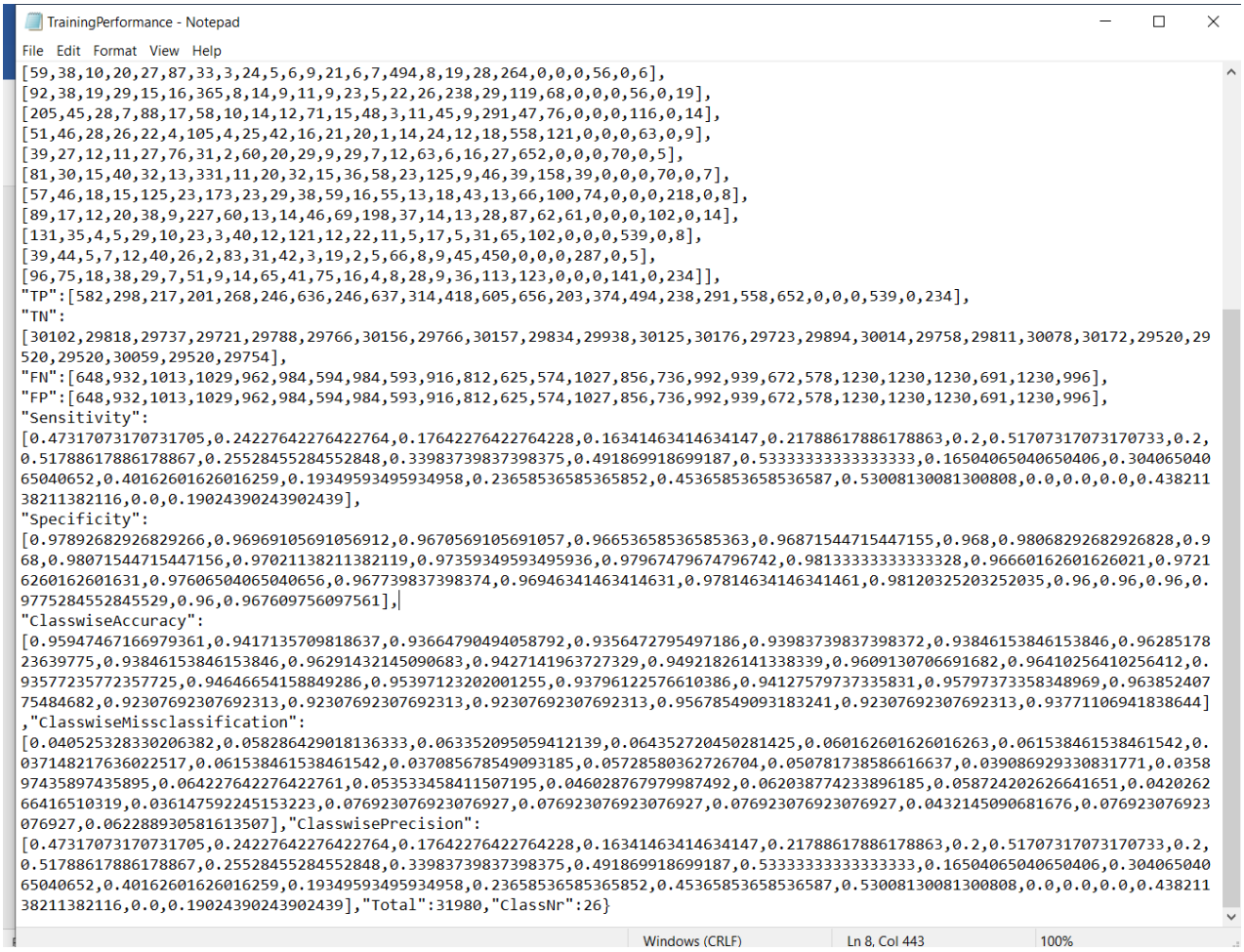


Confusion matrix for the training set.



Confusion matrix for the validation set

Performance JSONs



Performance JSON for the training test (the JSONs for the testing and validation sets are similar)

Future improvements:

While I am rather confident about the feed-forward section of the network, the backpropagation section is still somewhat of a blur. The network suffers from exploding gradients when using softmax with a learning rate higher than 0.0001 and does not seem to be learning when using sigmoid. The accuracy obtained with a random model is usually around 3%. The highest accuracy I was able achieve was 28% before a NaN error was obtained (which is a common problem with CNNs, even with frameworks like tensorflow). The network is very slow and needs to be made more efficient. The use of optimizers will also help.