ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ

ΠΟΛΥΤΕΧΝΙΚΗ ΣΧΟΛΗ

ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

Οπτική Αναγνώριση Χαρακτήρων, με την χρήση Μηχανικής Μάθησης, σε ZedBoard πλατφόρμα

Διπλωματική Εργασία

Ανθιμόπουλος Θεολόγος

Επιβλέπων: Σωτηρίου Χρήστος

Βόλος έτος 2020-21

ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ

ΠΟΛΥΤΕΧΝΙΚΗ ΣΧΟΛΗ

ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

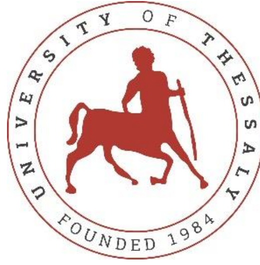
Οπτική Αναγνώριση Χαρακτήρων, με την χρήση Μηχανικής Μάθησης, σε ZedBoard πλατφόρμα

Διπλωματική Εργασία

Ανθιμόπουλος Θεολόγος

Επιβλέπων: Σωτηρίου Χρήστος

Βόλος έτος 2020-21



UNIVERSITY OF THESSALY

SCHOOL OF ENGINEERING

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

Optical Character Recognition (OCR), using Machine Learning, on the ZedBoard Platform

Diploma Thesis

Anthimopoulos Theologos

Supervisor: Sotiriou Christos

Volos year 2020-21

Εγκρίνεται από την Επιτροπή Εξέτασης:

Επιβλέπων **Σωτηρίου Χρήστος**
Αναπληρωτής Καθηγητής, Τμήμα Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών, Πανεπιστήμιο Θεσσαλίας

Μέλος **Πλέσσας Φώτιος**
Αναπληρωτής Καθηγητής, Τμήμα Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών, Πανεπιστήμιο Θεσσαλίας

Μέλος **Κατσαρός Δημήτριος**
Αναπληρωτής Καθηγητής, Τμήμα Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών, Πανεπιστήμιο Θεσσαλίας

Ημερομηνία έγκρισης: dd-mm-yyyy

ΕΥΧΑΡΙΣΤΙΕΣ ή ΣΧΟΛΙΑ

Ευχαριστώ τους γονείς μου και την θεία μου και τον αδελφό μου Κωνσταντίνο. Ακόμα θα ήθελα να ευχαριστήσω θερμά τον CAS lab του πανεπιστήμιου μου και ιδιαίτερα τον κ. Σωτηρίου για την συνεχή στήριξη και καθοδήγηση τους.

ΥΠΕΥΘΥΝΗ ΔΗΛΩΣΗ ΠΕΡΙ ΑΚΑΔΗΜΑΪΚΗΣ ΔΕΟΝΤΟΛΟΓΙΑΣ ΚΑΙ ΠΝΕΥΜΑΤΙΚΩΝ ΔΙΚΑΙΩΜΑΤΩΝ

Με πλήρη επίγνωση των συνεπειών του νόμου περί πνευματικών δικαιωμάτων, δηλώνω ρητά ότι η παρούσα διπλωματική εργασία, καθώς και τα ηλεκτρονικά αρχεία και πηγαίοι κώδικες που αναπτύχθηκαν ή τροποποιήθηκαν στα πλαίσια αυτής της εργασίας, αποτελεί αποκλειστικά προϊόν προσωπικής μου εργασίας, δεν προσβάλλει κάθε μορφής δικαιώματα διανοητικής ιδιοκτησίας, προσωπικότητας και προσωπικών δεδομένων τρίτων, δεν περιέχει έργα/εισφορές τρίτων για τα οποία απαιτείται άδεια των δημιουργών/δικαιούχων και δεν είναι προϊόν μερικής ή ολικής αντιγραφής, οι πηγές δε που χρησιμοποιήθηκαν περιορίζονται στις βιβλιογραφικές αναφορές και μόνον και πληρούν τους κανόνες της επιστημονικής παράθεσης. Τα σημεία όπου έχω χρησιμοποιήσει ιδέες, κείμενο, αρχεία ή/και πηγές άλλων συγγραφέων, αναφέρονται ευδιάκριτα στο κείμενο με την κατάλληλη παραπομπή και η σχετική αναφορά περιλαμβάνεται στο τμήμα των βιβλιογραφικών αναφορών με πλήρη περιγραφή. Αναλαμβάνω πλήρως, ατομικά και προσωπικά, όλες τις νομικές και διοικητικές συνέπειες που δύναται να προκύψουν στην περίπτωση κατά την οποία αποδειχθεί, διαχρονικά, ότι η εργασία αυτή ή τμήμα της δεν μου ανήκει διότι είναι προϊόν λογοκλοπής.

Ο/Η Δηλών/ούσα

(Υπογραφή)
Θεολόγος Ανθιμόπουλος
Ημερομηνία

ΠΕΡΙΛΗΨΗ

Σε αυτήν την πτυχιακή έχουμε υλοποιήσει ένα λογισμικό **Οπτικής Αναγνώριση Χαρακτήρων** σε ZedBoard πλατφόρμα. Η πρόκληση μας είναι να δημιουργήσουμε μια εφαρμογή που να είναι όσο το δυνατόν ποιο γρήγορη και να μπορεί να αναγνωρίσει μια ευρύ γκάμα εικόνων με αριθμούς και λατινικούς χαρακτήρες. Ακόμα ο χρόνος εκτέλεσης του θέλουμε να εξαρτάτε από την εικόνα, έτσι ώστε όσο το δυνατόν λιγότερες λέξεις έχουμε σε μια εικόνα τόσο ποιο γρήγορα να γίνετε η αναγνώριση. Για να το πετύχουμε αυτό, χρησιμοποιήσαμε μια προσέγγιση όπου οι χαρακτήρες εντοπίζονται και κατηγοριοποιούνται μεμονωμένα με τεχνικές **Μηχανικής Μάθησης**. Για να εντοπίσουμε τους χαρακτήρες χρησιμοποιήσαμε έναν αλγόριθμο που εξάγει τις εξαιρετικά σταθερές ακραίες περιοχές (**MSERs**) σε μια εικόνα. Ακόμα για να κατηγοριοποιήσουμε τους χαρακτήρες χρησιμοποιήσαμε ένα **Τεχνικό Νευρωνικό Δίκτυο** με semi-supervised προσέγγιση. Τέλος, το λογισμικό αυτό θέλουμε να το υλοποιήσουμε σε ZedBoard πλατφόρμα για να αξιοποίηση την **DPU** της, για ακόμα ποιο γρήγορα αποτελέσματα.

ABSTRACT

In this thesis we implement an Optical Character Recognition (**OCR**) application on ZedBoard platform. Our goal is to implement an OCR application that is fast enough, and can recognize a wide variety of images with Latin letters and numbers. Moreover, we want the time taken to perform an OCR task to depend on the input image, so that the fewer words we have in an image, the faster it may be recognized. To achieve this, we use an approach that detect and classify the characters in isolation with **Machine Learning** techniques. To detect the isolated characters, we use an algorithm that extracts the maximally stable extremal regions (MSERs) in an image. For the classification part we use an **Artificial Neural Network** trained with semi-supervised approach. Finally, we want to implement this application on ZedBoard platform to utilize its **DPU**, for even faster results.

REVISION HISTORY

ΠΕΡΙΛΗΨΗ .... *vii*

ABSTRACT .... *viii*

CHAPTER 1 .... *1*

INTRODUCTION *1*

1.1 Optical Character Recognition **19**

1.2 Aim of this Thesis **2**

1.3 Contribution of this Thesis **2**

1.4 Thesis Overview **4**

CHAPTER 2 .... *5*

Machine Learning Background *5*

2.1 Image Processing **5**

2.2 Artificial Neural Networks **6**

2.3.1. Multilayer Perceptron 6
2.3.2. Convolutional Neural Networks 6
2.3.3. Activation Functions 11
2.3.4. Batch Normalization and Pooling Layers 13
2.3.5. Dataset 14
2.3.6. Loss Functions and Optimization Algorithms 15
2.3.7. GANs 16

2.3 Object Localization **19**

CHAPTER 3 .... *22*

Xilinx ZedBoard Platform *22*

3.1 Xilinx ZedBoard FPGA **22**

2.3.1. ZedBoard PS and PL 24

3.2 Xilinx DPU **25**

3.3 Xilinx DNNDK **27**

CHAPTER 4 .... *28*

ZOCR APPLICATION OVERVIEW *28*

4.1 Introduction **28**

4.2 NN Model Creation **28**

4.2.1. Generator 29
4.2.2. Discriminator 31
4.2.3. Model Training 33

4.3 Character Isolation **35**

4.3.1.Introduction 35
4.3.2. MSER 36
4.3.3. Non-Maximum Suppression 39
4.3.4. ER Tracking 40

CHAPTER 5 42

ZOCR on ZEDBOARD PLATFORM 42

5.1 Introduction 42

5.2 NN Model Deployment 42
5.2.1 NN Model Compression 42
5.3.2 NN Model Compilation 46

5.3 Run OCRI on ZedBoard DPU 28

CHAPTER 6 50

EXPERIMENTAL RESULTS 50

6.1 NN Performance 50
6.1.1 NN Performance Comparison 52
6.1.2 DPU Performance Comparison 52

6.2 Character Isolation Performance 54

CHAPTER 7 56

CONCLUSION AND FUTURE WORK 56

REFRENCES 57

Chapter 1

Introduction

1.1 Optical Character Recognition

Optical character recognition (OCR) is the process of scanning handwritten or printed images and convert them into computer-identifiable texts. The first OCR invention was created in America in 1870 by Charles R. Carey [18]. This invention is an image scanner that uses a mosaic of photocells. After this invasion OCR technology starts to grow. The first product named “optophone” is developed in 1912 by Edmund Fournier d'Albe [19]. Optophone is a handheld scanner that when moved across a printed page, produces tones that corresponded to specific characters, so as to be interpreted by a blind person.

Nowadays, there are two most common methods that use Machine Learning techniques to solve this problem. The first method detects the characters in isolation, classify them, and then form them into words. So, this method contains tree steps, **character isolation**, **character classification** and **word formation.** The second method can perform the above steps with one model that is a Deep Neural Network. It is most used Deep Convolutional Recurrent Neural Networks (**DCRNN**) [20] because the convolutional layer works as character isolation and the recurrent layer works as character classification and word formation. In chapter 2 we will explain DCRNN in more detail.

Optical character recognition is a demanding process with a variety of applications. It can be challenging because the input image has complex background, varied lights intensity and large verity of colors. Thus, an OCR system with no errors is still challenging task even for modern OCRs. For this reason, in recent years OCR systems are becoming more efficient with specific types of input and can recognize images in a specific distribution, to maximize the accuracy of the system. They also use methods that are time consuming, and they need lot of computer resources. For example, one of the most known OCR application, Attention OCR [13] uses **1.3 billion** multiply and add operations to recognize an image. It will also work efficiently for images similar to French street names and cannot recognize other images e.g., “car plates”. This happens because it has been trained with the FSNS dataset [39], which contains street name signs cropped from Google Street View images of France, and has difficulties recognizing other image samples.

OCR accuracy can be increased if the output is constrained by a **lexicon** or with Natural Language Processing (**NLP**). Lexicon is a list of words that can correct the misspelling of the output. For example, attention OCR could use a list of French street names. Like a lexicon, NLP can transform the output of an OCR application to a form of human language using artificial intelligence. For this reason, attention OCR uses Recurrent Neural Networks [27] that can achieve this artificial transformation. But these techniques can be problematic, if we want to recognize a word that is not on this constrained number of words.

1.2 Aim of this Thesis

In this project we want to implement an OCR application for specific types of input. This software tool uses machine learning techniques such as neural networks. For the most efficient and fast use of the neural network, our goal is to implement this application in Xilinx ZedBoard platform [21] using its DPU. We want to detect and classify natural scene images like the ICDAR2003 dataset [1] shown in Figure 1. In order to simplify our application, we focus on images where the characters defer from the background and have fewer complex backgrounds.

OCRs are used in both business and industrial domain to reduce the time taken to scan and digitalize large documents or to solve automatization problems for IoT systems. Our aim is to create a system that is adaptable and can be used in many circumstances. For this reason, we will not use a lexicon or NLP techniques so that we can detect documents without a specific spelling layout. Finally, we want an efficient OCR system that can solve real problems in real time and can use limited resources.

1.3 Contribution of this Thesis

Contributions of the work that is described in this thesis can be summarized as follows:

- A state-of-the-art OCR application to recognize natural scene images is proposed. This presented method is developed on Xilinx ZedBoard platform to utilize its DPU.
- A Neural Network (NN) to classify natural scene images of characters is descripted.

This NN uses Convolutional Neural Networks to extract features and detect patterns to classify the character image.

- A training process for our NN with semi-supervised approach using a GAN is proposed. This method is trying to create samples that imitate characters using Artificial Intelligence. In consequence of this process our NN can understand characters from non-characters. The inspiration of this training technique was from the paper published by Tim Salimans at OpenAI [3].
- A machine learning algorithm that extracts possible character classes over an image is described. This algorithm is a part of OpenCV documentation [16] and computes the Maximally Stable Extremal Regions of an image. The name of the algorithm is “Linear Time MSER”, and it was proposed by Nistér D. and Stewénius H. [15]. Source code is also provided on OpenCV’s GitHub repository [5].
- A proposed methodology to convert the NN model to a quantized model that can be executed on Xilinx ZedBoard DPU [43]. To achieve this, we use Xilinx DNNDK tool [4] that converts the pure data and the structure of NN to a file that is the abbreviation for executable and linkable format and defines the structure for binaries, libraries, and core files.

Figure 1. Sample of the ICDR2003 Dataset [1].

1.4 Thesis Overview

This thesis is divided into six chapters. The current chapter have already provided the aims and the contributions of this thesis. The remaining chapters can be summarized as follows:

- Chapter 2 contains all the Machine Learning background needed to understand this thesis. It contains some basic image processing and object localization techniques, and an introduction to Artificial Neural Networks.
- Chapter 3 contains an introduction to ZedBoard platform. The contribution of ZedBoard platform to this thesis is also mentioned.
- Chapter 4 explains our based OCR approach method.
- Chapter 5 contains accuracy and performance measurements of our proposed OCR application. A performance comparison of the proposed method with existing state-of-the-art methods is also provided.
- Chapter 6 draws the collusions of our work and our future plans.

Chapter 2

Machine Learning Background

2.1 Image Processing

In this sub-chapter we are going to see some basic image conversion techniques used in this application. Image processing is a mathematical equation that can be applied to an image to convert it from one domain to another. Converting an image from one domain to another can enable the identification of features that may not be as easily detected in the previous domain.

An **RGB** image is a two-dimension 3-D array. We call each 3-D element of this array a pixel and represents the tree hues of light (**R** = red, **G** = green, **B** = blue). The pixel values of an image ranging from 0 to 255. The pixel (0,0,0) represents the white color and the (255,255,255) the black color. Changing the values of the three hues of light can create different colors.

A **Grayscale** image is a two-dimensional 1-D array. Each pixel is a simple sample that represents only an amount of light carrying the intercity. Grayscale scale is a black-and-white image with size (Cols, Rows, 1). A gray image is the average of the tree hues of light ($Gray = (R + G + B)/3$).

YCrCb [28] is a family of color spaces used as a part of the color image pipeline in video and digital photography systems. As shown in figure 2, it contains three channels the Luma and the Chroma (Cr, Cb). Luma represent the brightness in an image and Cb, Cr are the blue-difference and red-difference chroma components. The mathematical equation for each Y, Cr, Cb is (R, G, B is the tree hues of light):

$$Y = 16 + 65.738 * \frac{R}{255} + 129.057 * \frac{G}{256} + 25.064 * \frac{B}{256}$$

$$Cb = 128 - 37.945 * \frac{R}{255} - 74.494 * \frac{G}{256} + 112.439 * \frac{B}{256}$$

$$Cr = 128 + 112.439 * R - \ 94.154 * \frac{G}{256} - \ 18.285 * \frac{B}{256}$$

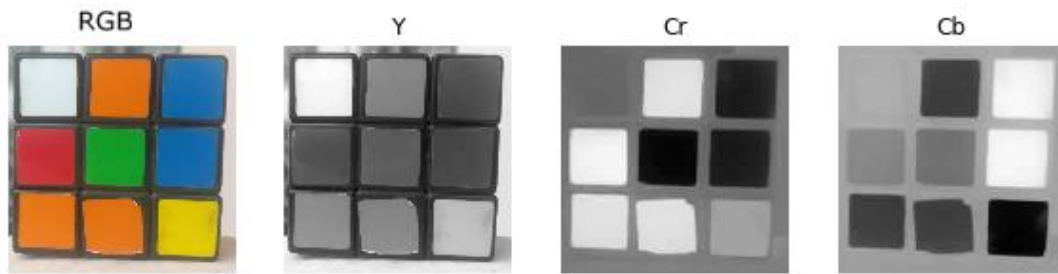


Figure 2: Luma (Y), and Chroma (Cr, Cb) Channels in YCrCb color space [28]

The **Sobel** [30] filter that also called Sobel operator, is an image transformation used in Machine Learning for edge detection algorithms. It was proposed by Irwin Sobel and Gary Feldma in 1968. Sobel creates an image emphasized in edges like the sample in Figure 3. In **OpenCV** documentation [29] we can see the mathematical formulation of Sobel operator with the theory. OpenCV is an open-source Computer Vision library that contains a lot of useful tools and libraries used in this thesis for image processing.

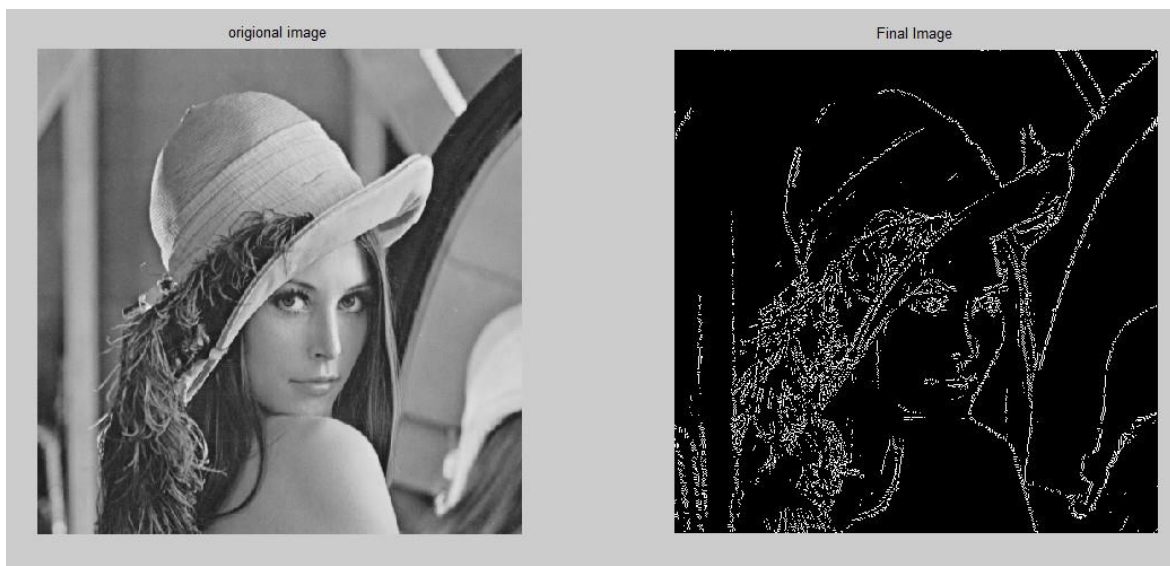


Figure 3: Sobel filter example. [29]

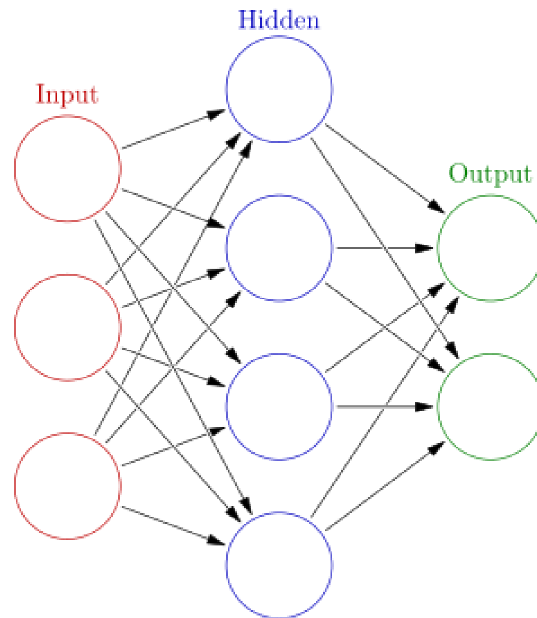
2.2 Artificial Neural Networks

2.3.1 Multilayer Perceptron

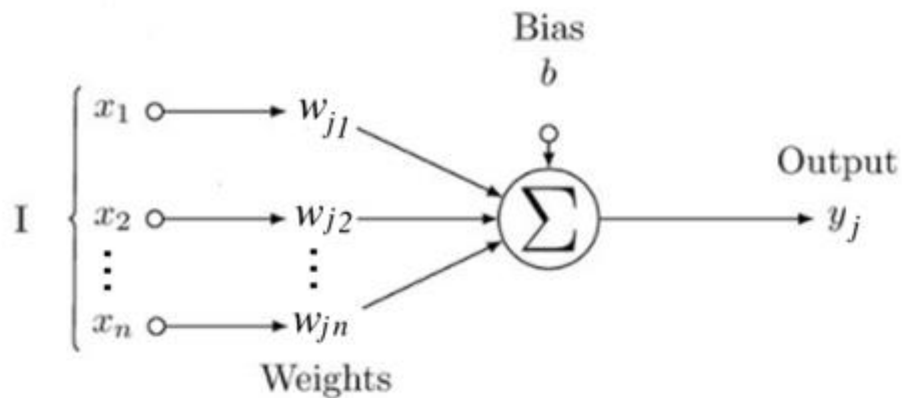
Artificial neural networks (**ANNs**) [24], usually simply called neural networks (**NNs**), are system models which map a given input to an output. Their name stems from their

supposed semblance to biological neural networks of the brain. The first NN architecture was proposed by F. Rosenblatt in 1958 [23]. We can see an example in figure 4.

Figure 4. Dense layer NN [14].



This NN is called Multilayer Perceptron (**MLP**), or Dense layer and it consist of multiple layers the input layer, the output layer and, the hidden layer. The output of a MLP corresponds to a probability for all the known classes. The output (y_j) of the *j* element of a layer, also called **gradient** or **node**, is illustrated by this Figure:



Input I, which is a vector, is multiplied to a vector called **weights** and summed and then is added a number called **bias**. In a layer, the result of Y *n*-dimensional vector can be described as a matrix multiplication as follow:

$$\vec{Y} = \begin{pmatrix} W1{,}1 & \cdots & W1,n \\ \vdots & \ddots & \vdots \\ Wn,1 & \cdots & Wn,n \end{pmatrix} * \vec{I} + \vec{B}\text{ias}$$

In each layer, weights and biases must have the appropriate values (float 32-bits) to process the given input signal correctly. The process of adjusting the weights and biases is called training. Training identifies appropriate parameters which benefits the model using functions that calculate its loss and tries to minimize it.

The input of a dense NN is a normalized vector with a certain size which its values must be decimal. When we create such an NN there is pre-process step that we transform each input to this desired vector. If we have a **classification problem,** for example RGB images of birds to determine their type, to transform this image to the appropriate input we must do the following pre-process steps:

- Transform images from RGB to grayscale to reduce dimensionality.
- Resize each image to a certain size e.g., 128x128x1 (size of input layer).
- Divide each image-matrix with 255 (Max gray pixel).
- Convert matrix to vector

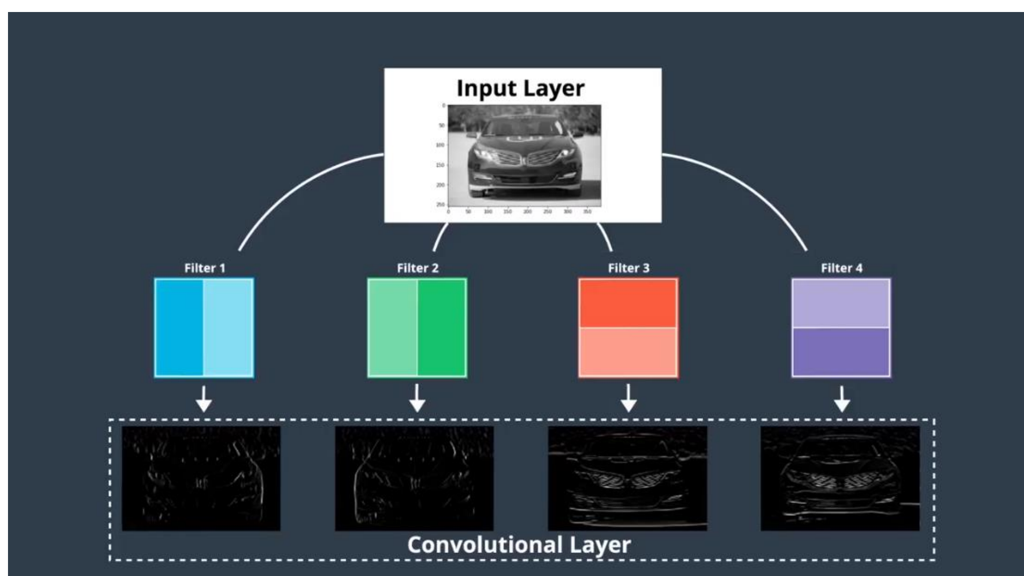
2.3.2 Convolutional Neural Networks (CNNs)

Consider this image of a dog sown bellow. There are many different patterns that we want to detect, for instance the region under its nose has teeth some whiskers and a tongue. To understand this image, we need filters for detecting all three of these characteristics one for each of teeth whiskers and tongue. To achieve that we use a **convolutional filter** under this specific area. Adding some other convolutional filters will extract other regions of the area and will give as more information. This set of convolutional filters is called **Convolutional layer**. A Convolutional layer has its own shared set of weights that different from the others. In fact, it is common to have 10s of hundreds of these collections in a convolutional layer each corresponding to their own filter.

Below in Figure 5 we can see the results if we apply an image of a car through four convolutional filters. We called these results "**feature maps**". When we visualize this feature maps, we see that they look like filtered images. We have taken all the complicated depth information in the original image and each of these four cases the output is a much simpler image with less information. By picking up the structure of the filters we can see that the first two filters discovered vertical edges were the last two detect horizontal edges in the image. The output of this Convolutional Layer is a stuck of four 2-dimensional arrays.

In practices it is most used a pipeline of Convolutional layers or CNNs. The idea is that each of the feature map in the first layer is used as input to another Convolutional layer to discover patters within the patters we discover in the first layer. The number of CNNs, which defines the architecture of our NN depend on the problem. Finally, we use a dense layer to classify these discovered features extracted by the Convolutional layers. Such a model is called Deep Convolutional Neural Network (**DCNN**) [40] and it used for classification problems.

Figure 5. Convolutional Layer.



Convolutional layers are not too deferent from the dense layers. Dense layers are fully connected meaning that the nodes are connected to every node in the previous layer. Convolutional layers are globally connected where their nodes are connected to only a small subset of the previous latest notes. In both cases they have weights and biases which in train process we try to find the most appropriate values. In the case of CNNs where the weights take the form of convolutional filters, those filters are randomly generated and so are the patterns that initially designed to detect. In train process we determine what kind of patterns it needs to detect based on a loss function. For instance, if the dataset contains images of dogs, the CNN can learn on its own filters that is able to detect the characteristics of a dog. In the above figure we can see a computational example of the first element of a feature map. As we can see that **Kernel size** is 3x3, which refers to the size of the filter. Filter is applied to an area of a matrix, and the output result is the sum of this pooled area as in the example. In a convolutional layer we must specify the **Strides.** This parameter represents the number of input shifts over the input matrix. In the above example the number of strides is 1.

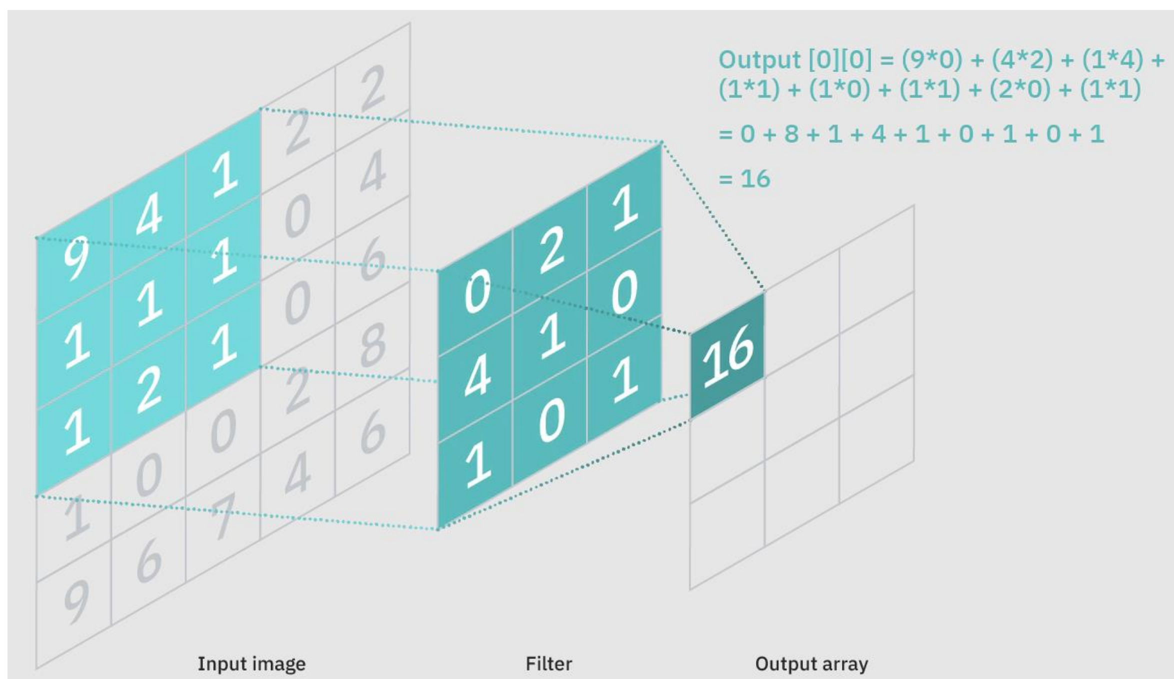


Figure 6. Convolutional filter calculation.

2.3.3 Activation functions

In NNs an activation function performs a specific mathematical operation on a node. In this thesis we use five action functions, **Sigmoid**, **Tanh**, **RELU**, **Leaky RELU** and **SoftMax.**

Sigmoid and **Tanh**
The sigmoid activation function σ(x) squashes a real-valued number into the range between zero and one as in:

$$\sigma(x) = \frac{1}{(1+e^{-x})}$$

On the other hand, tanh(x) squashes a real-valued number into the range between negative one and one as in:

$$tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

In figure 6 we can see the graphic representation of sigmoid and tanh. If the input vector was in the scale of [-1,1] sigmoid would be problematic because it will hide the negative information of the input and so tanh would be more appropriate. Sigmoid is used when the input is in the scale of [0,1]. It will hide the negative information that is not needed in the output. In practice a Sigmoid and Tanh are used in two-class classification problems. In our application we use *σ(x)* in the final output node of our NN to determine if a given input is a character or not.

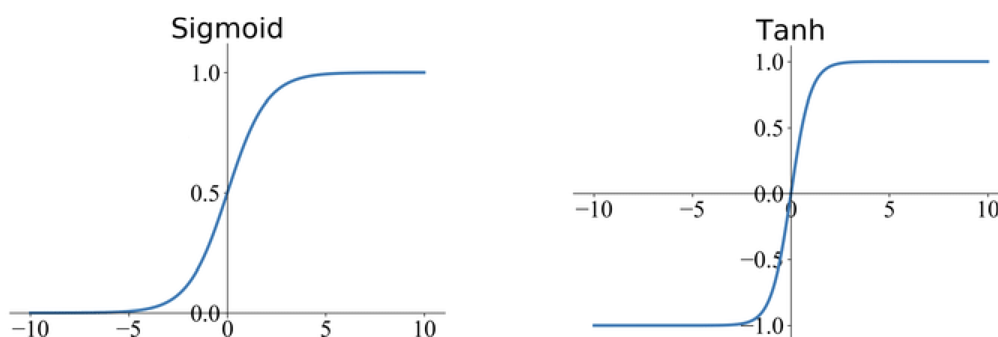


Figure 6. Sigmoid and Tanh graphic representations.

SoftMax activation function is used in Machine Learning for Multi-class classification problems that also called Multi-class Logistic Regression. In our application we use this function in the final output node of our NN to determine the character over all

known character classes. We can see an example of SoftMax in Figure 8. In the example the output of a dense layer has this form of a vector. In statistics terminology this output is called logits. SoftMax will determine with a probability of 90% that the output is zero character. Given a n-dimensional vector *X* the SoftMax activation function $S(x_i)$ squashes a real-valued number *Xi* into the range between zero and one as in:

$$S(x_i) = \frac{e^{x_i}}{\sum_{j=0}^{n} e^{-x_j}}$$

Rectified Linear Unit (**ReLU**) activation function is a linear and simply threshold at zero and can therefore be expressed as:

$$\mathrm{ReLU(x)} = \max(0, \mathrm{x})$$

Leaky ReLU activation function is also a simple threshold, allowing allow a small positive gradient when the unit is not active as:

$$\mathrm{Leaky\ ReLU(x)} = \max(\mathrm{x} * \alpha, \mathrm{x})$$

Parameter *α* is a small percentage value typically less than 10%. In practice, both of this units have one major drawback that arises from their simplicity. They can be very fragile during training because of their small gradient (output of the layer) when x < 0. A large gradient flowing through a neuron could cause the weights to update in such a way that the neuron will never activate on any datapoint again. If this happens, the gradient flowing through the unit will forever be zero. This phenomenon is referred to as dead neurons in the neural network context [32]. To avoid dead neurons, such functions are usually used in the output of a Convolutional Layer.

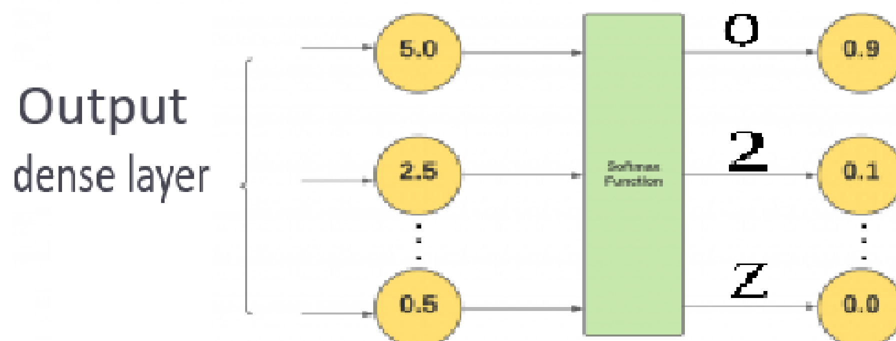


Figure 8. SoftMax example.

2.3.4 Batch Normalization and Pooling layers

Batch Normalization (**BN**) [37] was proposed by Sergey Loffe and Christian Szegedy to make NN models more stable during training. This could benefit the model by decreasing the time of training, and in some cases the efficiency of the model is also increased. At a hidden layer with an output vector *Z,* BN first calculates the mean (μ) and the standard deviation (σ) of a **batch**. Batch is a sample of the dataset to work with, before updating the internal model parameters. The normalized output Z_{norm} of BN is:

$$Z_{norm} = \gamma * \frac{Z - \mu}{\sigma} + \beta$$

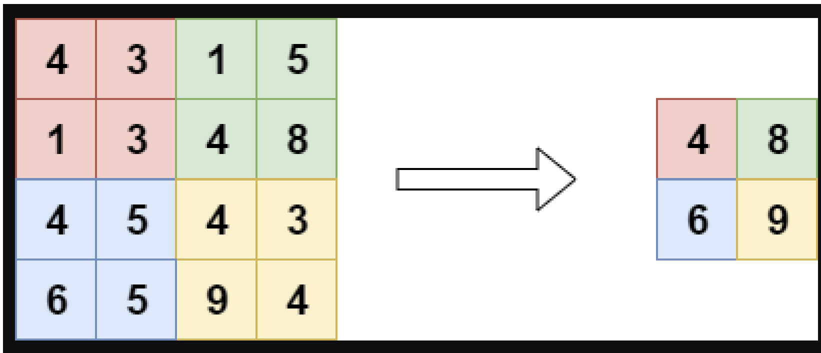
The γ and β are trainable parameters which allow to adjust the standard deviation and the bias, respectively.

In this thesis we use two deferent pooling layers, **Max pooling** [38] and **Average pooling** [38]. Pooling layers have the ability to reduce the dimensionality of a feature map usually after a CNN. This technique is referred to as downsampling of the feature map. Given a feature map, Max pooling fist involves computing the max value, like this:

4	3	1	5
1	3	4	8
4	5	4	3
6	5	9	4

Max([4, 3, 1, 3]) = 4

In the max pooling we must specify a window size which determines the area that we want to pool from the feature map. In the above example the window is 2x2. To downsample the feature map we must also specify the strides i.e., steps during the sliding operation. In the above example we perform Max pooling with window 2x2, and strides are 2.



As Max pooling is calculating the Max of a picked area, Average pooling is calculating the average.

2.3.5 Dataset

Neural networks learn (or are trained) by processing examples, each of which contains a known "input" and "result or labels," forming probability-weighted associations between the two, which are stored within the data structure of the net itself. This process is called **supervised learning**. The ability to learn is a peculiar feature pertaining to intelligent systems, biological or otherwise. In artificial systems, learning is viewed as the process of updating the internal representation of the system in response to external stimuli so that it can perform a specific task. This includes modifying the network architecture, which involves adjusting the weights of the links, pruning or creating some connection links, and/or changing the firing rules of the individual neurons (Schalkoff, 1997) [10]. NN learning is performed iteratively, as the network is presented with training examples, similar to the way humans learn from experience. For example, in image recognition, they might learn to identify images that contain cats, by analyzing example images manually labeled as "cat" or "no cat", and use the model to identify cats in other images. In this example the known classes are the "cat" and the "no cat". Thus, network output would represent the probability distribution of those two classes. In our application, we need to identify or classify characters, and we trained our model with the **Chars74k** dataset [9]. In figure 9, a small sample of the dataset is illustrated. Chars74k dataset contains 7704 characters obtained from natural images with 62 classes (0-9, A-Z, a-z). In order to not only classify but also to determine if the given input is a character, we use a **GAN** with semi-supervised approach. In the next sections we will explain in more detail the idea behind this approach.

The other dataset that we use is ICDAR2003 dataset [1]. This dataset contains three sub-datasets the Robust Reading and Text Locating (RRTL), Robust Character Recognition (RCR), and the Robust Word Recognition. We want to validate our application's accuracy in this dataset using the first two sub-datasets. The first sub-dataset contains images which we show later in Figure 1. The RCR contains images like the Chars74k dataset. This character images have been extracted from RRTL and labeled. RCR will help ass calculate the validation and test accuracy of our NN on another distributed data.

Figure 9. Project graph [9].

2.3.6 Loss functions and Optimization algorithms

The most popular **loss function** for image classification in NNs is the **cross-entropy loss**, generalized to multiple classes via an activation function **ϕ(v)** and the negative log likelihood. Mathematically, the cross-entropy loss of v has the form:

$$Cross_{entropy_{loss}}(\nu) = -\log(\varphi(\nu))$$

A sigmoid cross-entropy loss function is the negative log likelihood of *σ(x)* and SoftMax cross-entropy loss function is the negative log likelihood of *S(x)*.

In training when we process a labeled data from the dataset, NN will output a prediction in the form of a vector. We want after the activation function a probability value near 1 in the right predicted class and near zero in the others. To make our NN learn from the dataset, for each labeled data we calculate the loss of the right predicted class, and we try to minimize it.

To minimize the loss of the model we use optimization algorithms. One of the most know optimization algorithms is the **Adam Optimizer** [33]. Adam was proposed by Diederik Kingma from OpenAI and Jimmy Ba from the University of Toronto in 2015. It has the ability

to update NN’s weights and bias during training to increase the learning efficiency of the model. This process of adjusting weights and biases to benefit the models is called “Back propagation”.

2.3.7 GANs

A general adversarial network (**GAN**) [2] is a machine learning generative model able to generate new thinks. For example, new images that are realistic even though they have never been seen before. GANs, along with several other kinds of generative models, are using a function called differentiable function that can generate new sample that imitate a given dataset. In GANs this function is a NN called **generator**. Generator network takes random noises as input, to transform and reshape it to have a recognizable structure. The output of the generator network is a realistic image. The choice of random input noise determines which image will come out of the generator network. Running the generator network with many different input noise values produces many different realistic output images. The goal is for these images to be fair samples from distribution over real data. One application of GANs is to generate faces of anime characters. Yanghua Jin and others published a grate paper with title “Towards the Automatic Anime Characters Creation with Generative Adversarial Networks” [41] which explain with detail a GAN. Suppose that we have a dataset with anime characters like the sample in figure 10, and we want to create new anime characters.

Figure 10. Anime characters sample.

Yanghua Jin’s paper proposed a method to create a generator network that can create this anime characters like the example in figure 11. We can see that generator network takes as input a random vector with certain size and generates a new anime

character that does not belong to the dataset. Also, we can see that if a small change will be made in the random vector, generator will automatically be triggered to create another sample. For example, changing only the value of the fist vector of figure 11, from 0.1 to 3 will create an anime character with longer hair than the previous one.

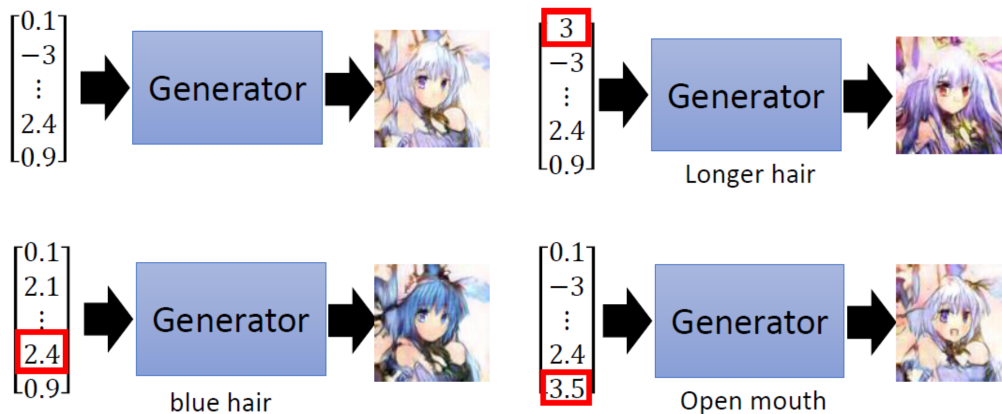


Figure 11. Generator for anime characters [41].

Training process for a GAN is different from the training process for a supervised learning model. In supervised learning we show the model an image of a traffic light and we tell the NN this is a traffic light. For a generative model, there is no output associated with each image, we show the model a lot of images, and ask it to make more images which come from the same probability distribution. To maximize the probability that the generator net will generate good images labeled to the training dataset, we use a second network called the **discriminator** which is a regular classifier. In training, discriminator is shown images from the dataset (**real images**) in half of the time and images created from the generator (**fake images**) the other half of the time. The discriminator is trained to output the probability that the input is real, so it tries to assign a probability near to one for real images and a probability 0 for fake images. This training process is called **Unsupervised Learning**. In figure 12 we can see an example of the discriminator network used in Yanghua Jin's paper. In the first two images which come from the dataset the probability value is 1. These images are of a very good quality, and discrimination determines that are real. The third image which is of a very bad quality, is generated from the generator, and has a low probability value (0.1) to be real image.

Figure 12. Generator for anime characters [41].

In training, generator tries to do the opposite from the discriminator, it is trained to try to output images to which the discriminator will assign a probability near 1 i.e. the input is real. Overtime the generator is forced to produce more realistic outputs to ''fool'' the discriminator, and the discriminator can learn to recognize if an image is a class of the dataset. In figure 13, we can see a GAN representation during training for three iterations (V1,2,3). Generator takes as input the random image and generates the new fake image and discriminator takes as input both images one at a time and determine if the image is real or fake. We can see that generator is improved during training creating more realistic samples. That also means discriminator is improved to identify which images is produced from the generator and which image comes from the dataset.

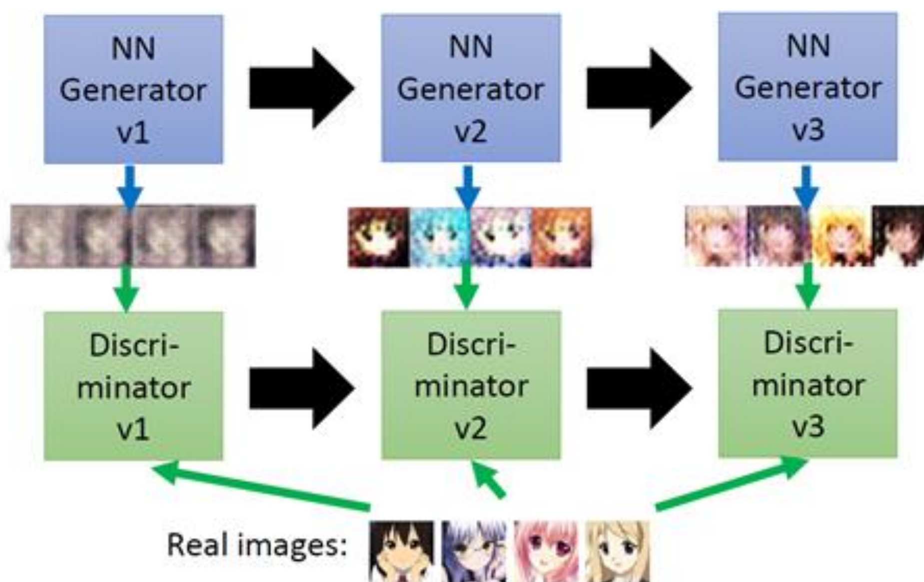


Figure 13. GAN training representation [41].

2.3 Object Localization

There are a lot of models and algorithms that localize objects or even characters from a given image. An **object localization algorithm** produces a list of object categories present in the image, along with an axis-aligned bounding box indicating the position and scale of one instance of each object category [25]. Such an algorithm is used in a wide variety of tasks such as scene detection, stereo matching, and object tracking for detecting the desired object. An example is an algorithm that computes the Maximally Stable Extremal Regions (**MSER**) of the source image. The idea is that MSER tries to detect common pixels in an image that might be an element. Like the human eye, when we read, we detect the letters from the background by understanding that a character has common pixel values and color intensity, creating an object with a certain structure (common height etc.).

The basic idea behind MSER is that, If we wanted to locate the "F" letter in figure 14 and we knew that "F" is white, we could easily curve join the continuous points having the white color. If we do not know the color of "F" we have to search for continuous points in different pixel intensity values.

Figure 14. Detect "F"

Another model that can locate objects from an image is a CNN. CNNs can be trained to identify that an area with common pixel values and color intensity with a certain structure, can be an object. In figure 15 we can see how CNNs locate the letters from an

image. The output image of CNN is the feature map of the last Convolutional layer. We can see that in the feature map, CNN discovered handwritten letters and emphasize all the useful information with white pixels.

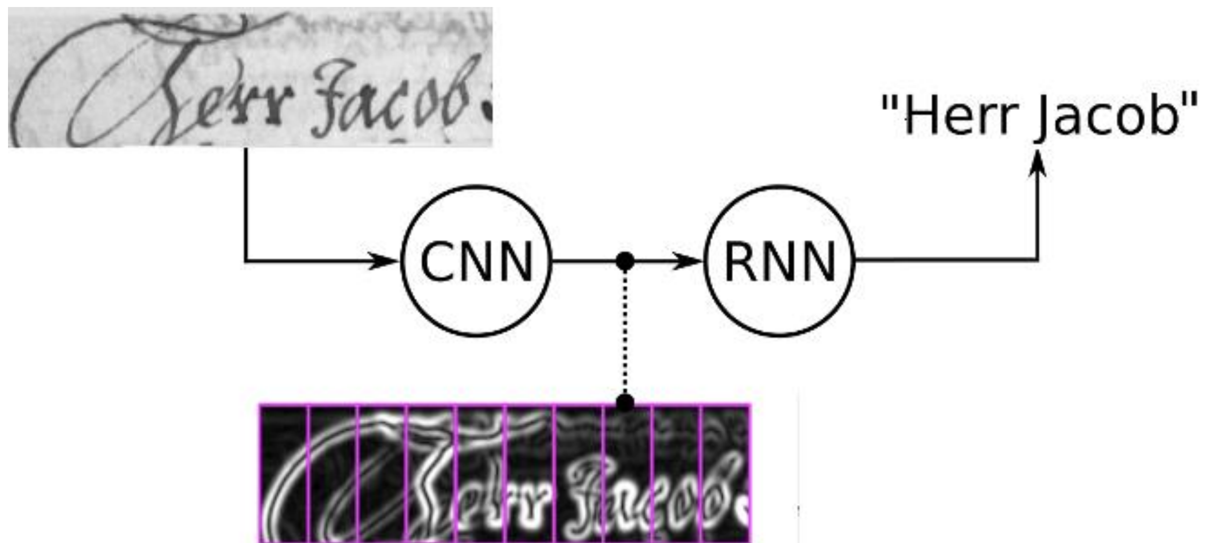


Figure 15. CNN for object localization [26].

Most OCR applications use CNNs as an encoder to detect letters. MSER detects the desired objects along with a few outliers, as distinct from CNNs that detect only the desired objects. To perform OCR, we apply to CNNs a decoder model that will classify the detected objects. In most cases this decoder is a Recurrent Neural Network (**RNN**) [27]. The detected characters are fed into the RNN one at the time, and the network is trigged to predict a character or a sequence of characters. RNNs and dense layers have a similar general structure but also have a difference. In the dense layer the output at any time is a function of the current weights and the bias. As we can see in figure 16, in RNNs the output at *time t* depends on the current input the weights and the bias but also on previous inputs such as the *t-1, t-2* etc*.* For example, the correct classification of “r” in the sixths bounding box of feature map, in figure 13, depends on the first five bounding boxes.

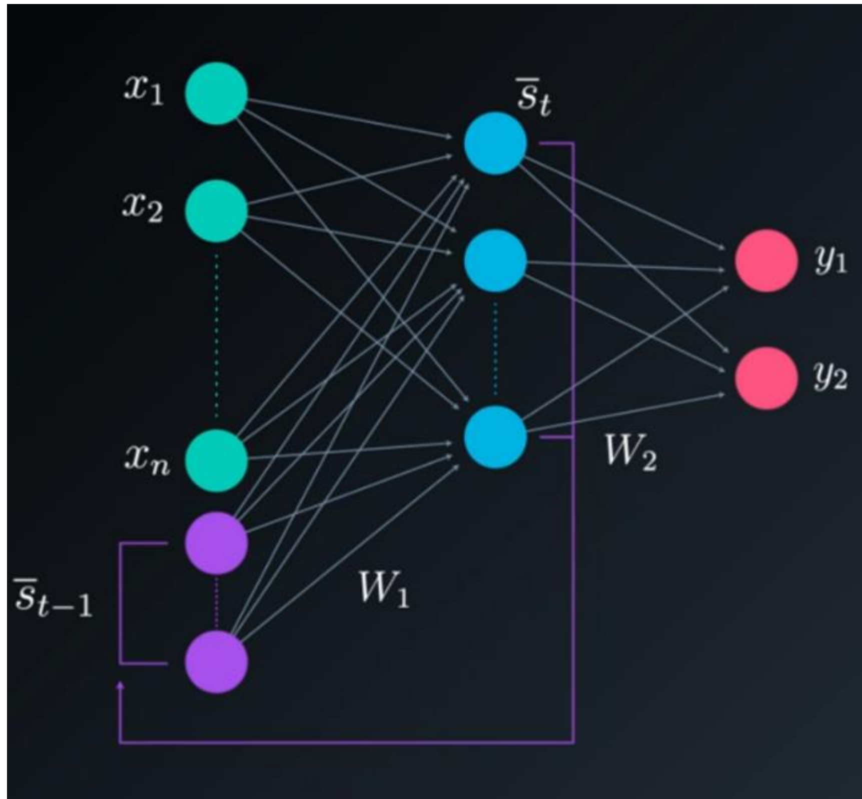


Figure 16. RNN basic architecture [27].

Chapter 3

Xilinx ZedBoard Platform

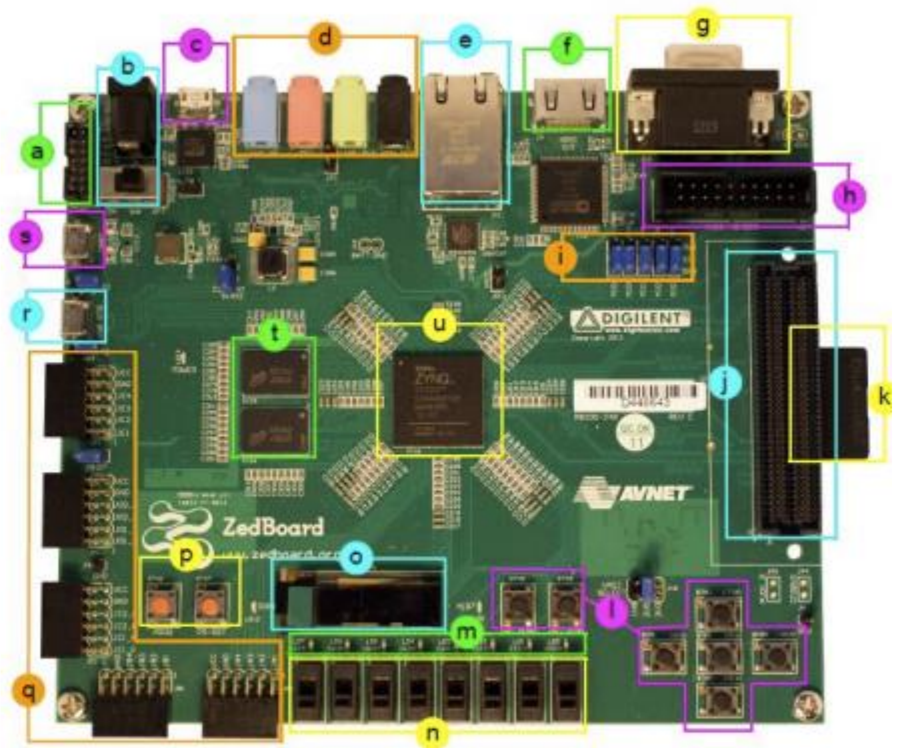
3.1 Xilinx ZedBoard FPGA

ZedBoard [21] platform is a Xilinx FPGA (Field Programmable Gate Array) board, designed to be programable and configurable by the circuit designer. FPGA is an application-specific integrated circuit (**ASIC**) that contains a series of hardware components like Flip-Flops (**FFs**), Multiplexers, Look-Up Tables (**LUTs**), Digital Signal Processors (**DSPs**), Block RAMs (**BRAMs**) and CPU. FFs are small elements of gates able to store one data bit between cycles. Multiplexer is a circuit devise that forwards a chosen signal between several signals, based on a selector input signal. LUT is an N-bit table of pre-defined responses for each unique set of inputs. DSPs are block units that contain adders, subtractors and multipliers. A BRAM is a block of single/dual port RAM memory to store an amount of data. FFs, Multiplexers, LUTs and other logic components are combined to create configurable logic blocks (**CLBs**). CLB is the leading resource containing the design logic in FPGA and the main functionality in logic design. The way to configure and utilize this CLBs is to use a hardware description language (**HDL**) like Verilog, VHDL etc. [42]. CLBs can combine for more complex operations such as multipliers, registers, counters and even DSP functions.

CLBs gives ASIC designers the flexibility to develop, simulate, and run modeling routines to ASIC prototypes onto the FPGA. Such a flexibility is difficult or impossible to achieve with an ASIC. FPGAs because of their amount of computer resources has application in machine learning algorithms, AI models, hardware accelerators, wireless communications, and others. In the next sub-chapter, we are going to describe the contribution of ZedBoard in machine learning specifically for our application. Because we implement our OCR in ZedBoard, we call our application **ZOCR**.

ZedBoard supports the implementation of operating systems such as Linux, Windows and Android or other operating system-based design system and Real Time operating system. One basic element of ZedBoard FPGA is chip Xilinx Zynq®-7000 All Programmable SoC. Zynq700 contains a dual-core CPU the ARM Cortex-A9. It also contains:

a) Xilinx JTAG Connector
b) Power input and switch
c) USB-JTAG (programmable)
d) Audio ports
e) 10/100/1000 Ethernet port
f) HDMI port
g) VGA port
h) XADC header port
i) Pog & reset buttons
j) FMC connector
k) SD card
l) User push button
m) LEDs
n) Switches
o) OLED display
p) Configuration jumpers
q) Pmod connector ports
r) USB-PTG peripheral ports
s) USB-UART port
t) DDR3 memory 512 MB
u) ARM CPU



ARM dual-core CPU is included in the Process Unit (**PS**). PS unit along with Multiplexed Input/Output (**MIO**) and Programmable Logic (**PL**), constitute the main structure of ZedBoard. MIO contains peripherals such as SD port, UART port, Ethernet port etc. PL contains the general-purpose FPGA logic fabric. It consists of:

- **Slices –** Slice is a subunit inside a CLB that used in combinatorial and sequential logic circuits. It consists of 4 LUTs, 8 FFs and other logic components.
- CLBs
- Input/Output Blocks (**IOBs**) – IOBs are a group of basic elements that implement the input and the output functions of FPGA. IOBs are the connection interface between the PL logic and the pads.

In figure 17 we can see the block diagram of ZedBoard. MIO connects PS to the outside world while PS and PL are combined to develop applications.

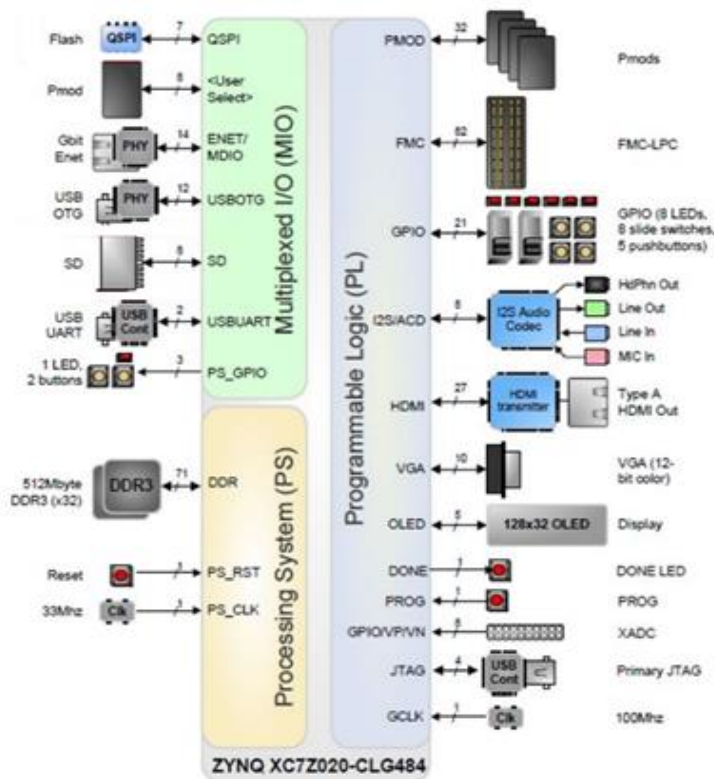


Figure 17. ZedBoard block diagram [21].

3.1.1 ZedBoard PS and PS

The basic feature of ZedBoard in the PS of Zynq is a set of components that create the Application Prosses Unit (**APU**). We can see in figure 18 a simplified graph of APU. APU is manly composed of two ARM CPU cores, where its core contains a functional block. Each block of the core contains a media processing engine (**MPE**), a floating-point unit (**FPU**), a two-partitioned level 1 cache memory (**L1 cache**), and a memory management unit (**MMU**). On-chip memory interconnector (**OCM**) along with, a SRAM, and a level 3 cache memory (**L2 cache**) are also included in APU. Communication between, the L1-chases of the cores, the L2 cache, the SRAM, and the OCM is accomplished though the snoop controller unit (**SCU**). PS is also composed of the following functional blocks:

- Memory interfaces
- I/O peripherals (IOP)
- Interconnect

The **PS I/O peripherals**, including the static/flash memory interfaces share a MIO of up to 54 MIO pins. Interconnect blocks are used from PS to communicate with PL via an Advanced

eXtensible Interface (**AXI)**. AXI is a Hight-Performance (**HP**) interface that is used from Zynq to create a gateway between PL and PS. The PS and PL can be tightly or loosely coupled using multiple interfaces and other signals that have a combined total of over 3,000 connections. This enables to effectively integrate user-created hardware accelerators and other functions in the PL logic that are accessible to the processors and can also access memory resources in the processing system.

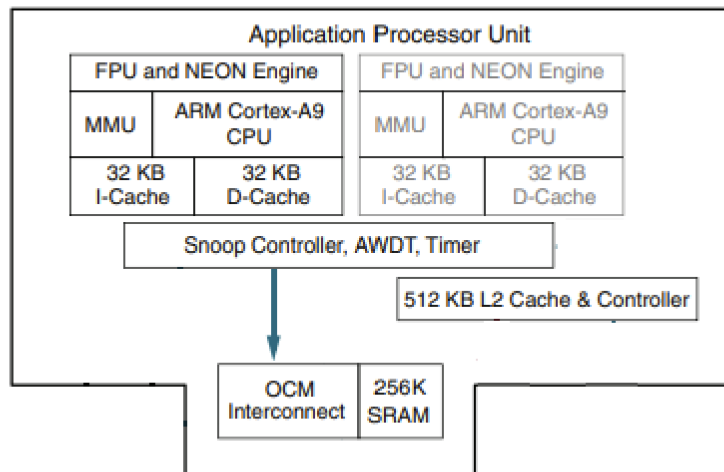


Figure 18. APU graph [21].

3.2 Xilinx DPU

Xilinx provides a library of Intellectual Property (**IP**) that contains a series of specialized subsystems, peripheral interfaces, and accelerators between PL and PS. In most cases, IP is developed in PL unit, to utilize the FPGA fabric, and is used from PS. PS provides the portability of the IP, while PL provides computational resources. For example, if we want to use SoftMax function to our application we could easily create and run a software implementation in the ARM CPU. Xilinx provides hardware implementation of SoftMax throughout an IP. The hardware implementation of SoftMax can be 160 times faster than a software implementation. The hardware SoftMax module takes approximately 10000 LUTs, 4 BRAMs, and 14 DSPs [21]. PS unit is used for data transfers and for interrupts. Interrupts are signals to the processor that trigger a response to an event that needs attention by the software. For example, an interrupt would be that SoftMax calculation is finished.

As we said in a previous chapter, NNs consists of layers (Convolutional, Dense, Recurrent) where the input is modified by some weights and summed with a bias, so it consists of multiply and add operations (linear operations). **Xilinx DPU** [43] is an IP unit that can perform linear operations across the layers, faster than CPUs and can process streams of data, which is essential for this project. In figure 19 we can see the block diagram of DPU. The DPU IP is implemented in PL of Zynq device with direct connections to PS via an AXI bus. A program running on the APU is also required to service interrupts and coordinate data transfers. DPU unit contains a high-performance scheduler module (**HPSM**), a hybrid computing array module (**HCA**), a global memory pool module (**GMPM**), and an instruction fetch unit. The HCA unit contains all computing resurfaces need to accelerate a NN layer. It is mainly composed of multipliers and adders. HCA requires storage for the input images as well as for the temporary and output data. GMPM which is linked with RAM unit via a high-speed data tube, will provide this memory locations. DPU also requires instructions to implement a neural network which are like the assembly of DPU. The instruction fetch unit will provide these instructions from APU, via a HPSM, to the computing resources of DPU.

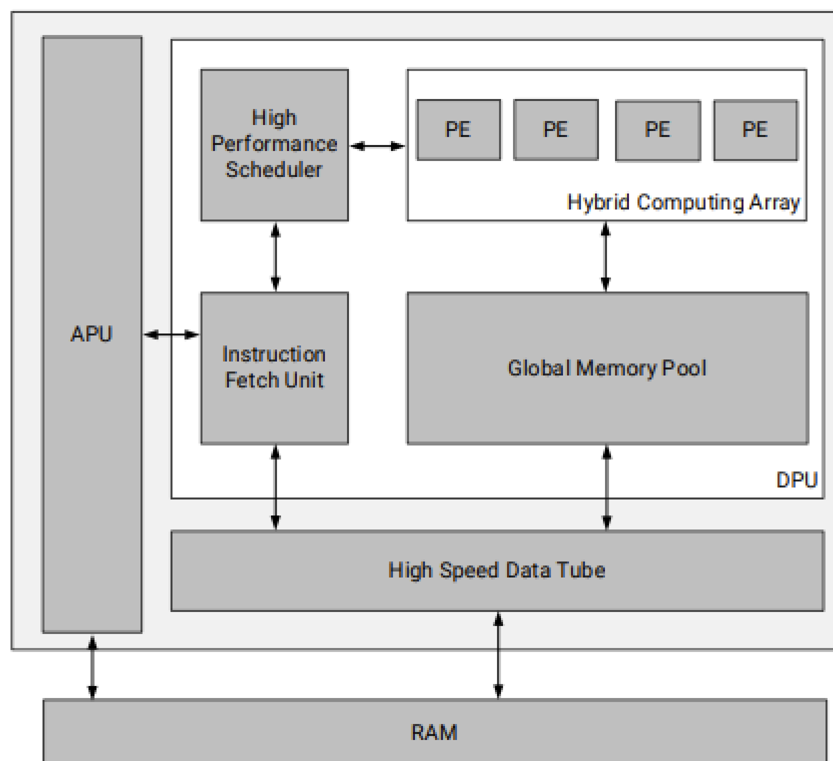


Figure 19. Xilinx DPU block diagram [43].

3.3 Xilinx DNNDK

The Deep Neural Network Development Kit (DNNDK) [4] is a deep learning toolchain for inference with the DPU. One of the basic DNNDK tools are the Deep Compression Tool (**DECENT**), and the Deep Neural Network Compiler (**DNNC**). These tools are installed in the host machine where we create our NN following the recommendation of Xilinx's documentation [4].

Xilinx gives us the capability to utilize DPU and optimize the trained model using the DNNDK tool. This tool takes as input the neural network file (.pd) that was created after training and maps it into a file (.elf) identifiable by Xilinx DPU. The ".pd" file is pure data and contains the weights and bias of the NN model along with its structure. The ".elf" file contains the instructions of DPU which are strongly related to the DPU architecture, the given NN, and the AXI Interconnect.

The computation unit inside the HCA which calculate the linear operations of the NN, contains multipliers and adders with 8-bit integers as input. But the NN in the ".pd" file contains linear operation with 32-bit floats. DECENT will optimize the weights and bias of the network from 32-bit floating point numbers to 8-bit integers. This optimization is called **quantization**. By converting the 32-bit floating-point weights and bias to 8-bit integer (INT8), the DECENT quantize tool can reduce the computing complexity without losing prediction accuracy. The fixed-point network model requires less memory bandwidth, thus providing faster speed and higher power efficiency than the floating-point model. This tool supports common layers in NNs, such as convolution, activation functions, dense, and BN. DECENT also provided TensorFlow [11]. TensorFlow is an end-to-end open-source platform for machine learning. Using this platform, we can create our NN by using TensorFlow extended libraries that can characterize, train, and extract NN architecture using python.

DNNC will take as input the quantized model and will create the ".elf" file. This file can be accessible into the board, by the developer, using DNNDK drivers with C++. Xilinx also include a Linux image that can boot ZedBoard and run C++ code on the processor. Linux image is a Linux operating system without graphical user interface (**GUI**). Accessibility to this Linux images can be accomplished through ZedBoard ethernet connection. DNNDK drivers also contain the appropriate version of OpenCV (3.4) that is essential for this project, for images processing tasks.

Chapter 4

ZOCR Application Overview

4.1 Introduction

In this chapter we present the design and implementation of ZOCR application. Our OCR implementation consists of two parts, **character isolation** and **character classification**. When we localize and recognize the characters from a given image and form them into words, OCR is complete. Both parts are written in C++, and they use a NN.

The algorithm first computes MSER of the source image using OpenCV source code. This algorithm joins the continuous points that have same color or intensity and returns the bounding boxes by identifying the start and the end point of each isolated box. These bounding boxes contain characters but also some outliers. The outliers can be small or big background noise, inside noise of a character or some pixel difference. In figure 20 we can see the results of this algorithm. In order to reduce the number of outliers we run an algorithm based on the fact that characters have a specific topology and structure, called **ER extraction**. This algorithm is part of character isolation which is converted in more detail in the next chapters.

The next step is to run our NN model for each detected bounding box. The model will give as two outputs. The first output will be about what letter-number that contour is (0-9, A-Z, a-z). The second output will be a probability number (0-1) that this contour is a character. Finally, we threshold those probability numbers and form characters into words using again some topology and structure techniques.

4.2 NN Model Creation

One way of improving the learning efficiency of deep learning models is **semi-supervised learning** [3]. In semi-supervised learning, the model can learn from labeled examples, but also gets better in classification by studying unlabeled examples even though those examples have no class labels. In Semi-supervised classification with GANs we set up the discriminator to work as a two-stage classifier.

GANs contain two models the generator and the discriminator, usually we train both and then use only the generator NN at the end of training to create samples.

Discriminator is usually of secondary importance and only used to train the generator. For semi-supervised learning we will focus on the discriminator. Discriminator is used to classify new data after we are done training and now generator becomes less important. The discriminator can not only learn from labeled data come from the dataset but also from unlabeled data come from generator. It contains two outputs, fist output for the labeled classes and second for the input image if it is real or fake. In our project the two-stage classification problem will be the 62 character classes, and if the given input is a character or not. We build a GAN with convolutional layers in the generator and the discriminator. This GAN architecture is called Deep Convolutional GAN (**DCGAN**).

Figure 20. Detected contours of an image with outlier

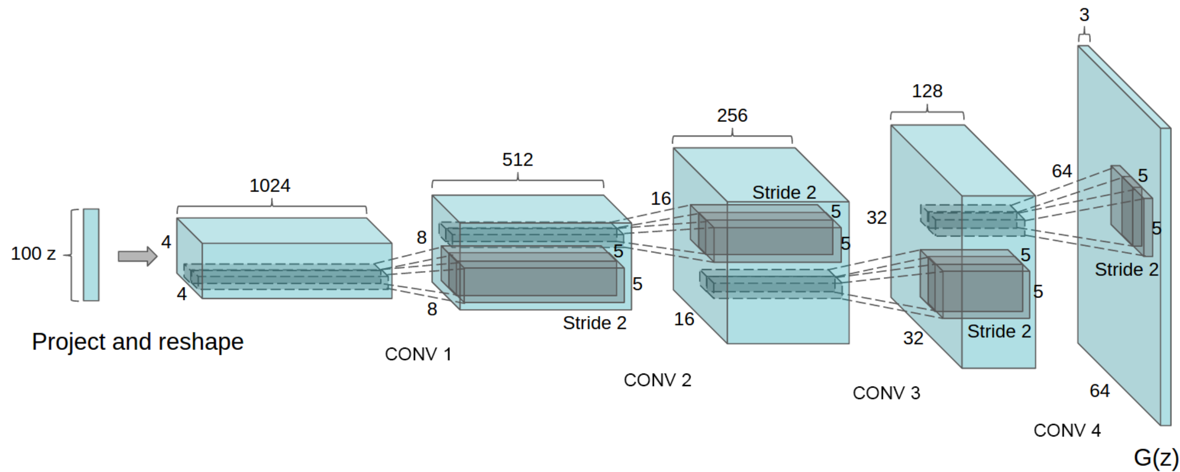
4.2.1 Generator

The generative NN that we use in our DCGAN was proposed by Alec Radford and his associates [2], and its architecture is shown in figure 21. The input of generator will be a random vector `z` with size 1x100. The output will be a *tanh* output, but this time with size 32x32x1 (not 64x64 as the paper) i.e., size of our Chars74K images.

The first made layer is a fully connected layer which is reshaped into a deep and narrow layer, 4x4x1024 as in the original DCGAN paper. Then we use batch normalization and a leaky ReLU activation function. Next is a transposed convolution layer where typically you would halve the depth and double the width and height of the previous layer. This

convolution transpose takes place in several layers as we want to get a larger and larger image with fewer and fewer channels until eventually, we have the desirable output image. Again, we use batch normalization and leaky ReLU.

Figure 21. Alec Radford generative model [2].



The presented model is used only for training our classifier. It can create images related to characters with an input `z` but not successfully enough. There are other papers that focus on the generator and can create acceptable or even identical samples like the one [41]. In Figure 22, we show a sample of generator output images in the last stage of training. We can see some fair sample of the dataset like images with 6, 2 or a. But there are also a lot of samples that are not recognizable or have a slight similarity with characters. Classifier will take as input this samples and it will be trained to understand that this are from generator. After trying classifier can determines the character samples from the non-characters.

Figure 22. Sample images created by generator.

4.2.2 Discriminator

The next step is to create our discriminator network. This network in train process takes as input half of the time images from the dataset and the other half from generator. Our goal in semi supervised learning is to make a good classifier that generalizes well for the test set even though we do not have many labeled examples. We are going to train the discriminator network, which is now a multi class model (more than an output). We use convolution and batch normalization, and we do not use any Max pooling [38] or average pooling [38]. We follow the recommendations from the paper by Tim Salimans at OpenAI with tittle "*Improved Techniques for Training GANs*" [3].

We do not use any batch normalization in the first three layers of the discriminator in order to have the correct mean and standard deviation as we get deeper in the network. We do use batch normalization and pooling to reduce the feature size. Batch normalization subtracts off the mean of every feature value and then adds on an offset parameter. Batch normalization goes ahead and sets the mean to be exactly equal to its bias parameter. That means at least feature values would all get set to just have that particular mean. We use dropout [36] a little bit more frequently here than some of the other models. The reason is that dropout helps to make sure that testing is not too much higher than the training error and reduce the amount of overfitting.

In figure 23, we can see the architecture of discriminator. This is slightly deferent from the architecture of OpenAI's paper. The purpose is to have a simple architecture in order to be faster and can be quantized by DNNDK tool. This model contains approximately **453K** trainable parameters (Float32) with input size 32x32x1.

After the creation of the discriminator, we must return its outputs. Remember that we have two outputs. The first output is the probability over all the real classes (62 classes of chars), and it is the SoftMax of the dense output (**class logits**). The second output is the **GAN logits**. We set that GAN logits to give as the probability that the input is character, such that:

$$P(input\ is\ char) = Sigmoid(GAN\ logits)$$

We need to transform class logits, which is multiclass SoftMax distribution, into a binary real-vs-fake decision that can be described with a sigmoid. Class logits is a 62-demational vector (62-classes) with log probability values. That means that its values may be very large, and either negative or positive. We need a function that can characterize GAN logits with all values of Class logits. With these ideas in mind, Tim Salimans proposed [3] the log-sum-exp (**LSE**) operation calculating the GAN logits such as:

$$GAN\ logits = \log\left(\sum_{n=1}^{62} e^{class_logits(n)}\right) = m + \log\left(\sum_{n=1}^{62} e^{class_logits(n) - m}\right)$$

This is numerical stable when $m = maximum\ (class_logits\ )$. This is because we ensure that the largest positive exponentiated term is exp(0) = 1.

```
Model: "sequential_12"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 15, 15, 64)        640
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 7, 7, 64)          36928
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 3, 3, 64)          36928
_________________________________________________________________
batch_normalization_1 (Batc (None, 3, 3, 64)          256
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 1, 1, 128)         73856
_________________________________________________________________
batch_normalization_2 (Batc (None, 1, 1, 128)         512
_________________________________________________________________
conv2d_5 (Conv2D)            (None, 1, 1, 128)         147584
_________________________________________________________________
batch_normalization_3 (Batc (None, 1, 1, 128)         512
_________________________________________________________________
conv2d_6 (Conv2D)            (None, 1, 1, 128)         147584
_________________________________________________________________
flatten_1 (Flatten)          (None, 128)               0
_________________________________________________________________
dense_1 (Dense)              (None, 62)                7998
=================================================================
Total params: 453,054
Trainable params: 452,286
Non-trainable params: 768
```

Figure 23. Architecture of discriminator with Keras.

4.2.3 Model training

The training process starts by identifying and preparing dataset. The Chars74k dataset [9] contains only 7705 characters obtained from natural images. This is quite small for training our classifier, so data augmentation helps as toward better generalization. To enrich our dataset, we transform each image 50 times by:

- Random rotation between -10 and 10 degrees.
- Random translation between -10 and 10 pixels in any direction.
- Random zoom between factors of 1 and 1.3.
- Random shearing between -25 and 25 degrees.
- Boolean choice to invert colors.
- Sobel edge detector filter [29] applied to 1/4 of images.

The new dataset is now 7705x50 after the data augmentation. Sobel filter along with the other applied filters create new versions of existing images that can prevent our generator to learn from a specific distributed dataset. Data normalization also applied to each image by scale pixel values to the range 0-1. This step is essential for our classifier to avoid overfitting.

The next step is to calculate the **loss** and to pose the **optimization operations** of the DCGAN. This step is the same as the paper at OpenAI. At the loss functions we first run our models. Firstly, the generator to generate new images and then the discriminator two times, for the real and for the fake classes. Then we compute the losses. For the discriminator, the loss function should combine two different losses:

1. The loss for the GAN problem, where we minimize the cross-entropy loss for the binary real-vs-fake classification problem.
2. The loss for character's classification problem, where we minimize the cross-entropy loss for the multi-class SoftMax.

The first part is the unsupervised part. In Figure 24, we show how this implementation in TensorFlow 1.12. The unsupervised part is divided into two different terms, real data loss and on fake data loss. We are dealing with a binary classification problem, so we use this **sigmoid cross entropy loss function**. In the discriminator, real loss calculation (**d_loss_real**), i.e., real data that comes from the dataset output should be all ones, because we want to say that all real data are characters. For the fake data (**d_loss_fake**), labels are all zeros, because we want to say all fake data coming from the generator are not

characters. The second part of the loss is the supervised portion of the semi supervised learning algorithm. It is used the cross entropy in terms of SoftMax between the labels for which class is present and the output over all the different classes (**class_cross_entropy**). The mean of the class_cross_entropy gives as the classifier's loss (**d_loss_class**). Finally, we add all those losses to calculate the discriminator loss (**d_loss**).

In the generator loss calculation, we use a loss function called **feature matching** that was embedded by Tim at open AI [2]. The basic idea of feature matching is that we take some features from a hidden layer of the discriminator and make sure that the average feature value on the training data is roughly comparable to the average feature value.

In statistics terminology when we take some statistics, from one dataset and from another dataset we ask those statistics to be similar. There is a learning technique called **Moment Matching** [34]**.** The use of the moment matching in GANs is described in the paper of Yujia Li, Kevin Swersky, and Richard S. Zemel with title "Generative moment matching networks". Each of the statistics that we extract is called a moment. According to Tim at open AI [2], the moments of generator are average values of features from the last convolutional layer of the discriminator.

First, we compute the moments of the dataset by taking the mean across all the features that we pulled out of the discriminator.

$$data\ moment = Mean(dataset_features\ )$$

Next, we compute the same moments in the same way but for the distribution of values that come from the generator (sample features) rather than from the training dataset. Finally, we compute the mean absolute difference between these two sets of moments, and we use that as the loss for the generator.

$$generator\ loss = Mean(abs(data_momets - sample_moments\ ))$$

That encouraged the generator to make sure that all the feature values of the discriminator have approximately the same average value regardless of whether the discriminator is run on the input or on generator's samples.

For the optimization operations we use **Adam optimizer** to reduce the loss of the generator and the discriminator. The learning rate is **0.001**. Learning rate is a hyperparameter that controls how quickly the model is adapted to the problem and

because of its importance it was calculated experimentally. We try lots of learning rate values to find the most appropriate.

Figure 24. Discriminator loss calculation with TensorFlow

```
 1 d_loss_real = tf.reduce_mean(
 2     tf.nn.sigmoid_cross_entropy_with_logits(logits=gan_logits_on_data,
 3                                labels=tf.ones_like(gan_logits_on_data)
 4                                              )
 5 )
 6 d_loss_fake = tf.reduce_mean(
 7  tf.nn.sigmoid_cross_entropy_with_logits(logits=gan_logits_on_samples,
 8                            labels=tf.zeros_like(gan_logits_on_samples)
 9                             )
10 )
11 y = tf.squeeze(y)
12 class_cross_entropy = tf.nn.softmax_cross_entropy_with_logits(
13       logits=class_logits_on_data,
14       labels=tf.one_hot(y, num_classes,dtype=tf.float32)
15                                                               )
16 class_cross_entropy = tf.squeeze(class_cross_entropy)
17
18
19
20 d_loss_class = tf.reduce_mean(class_cross_entropy)
21
```

4.3 Character Isolation

4.3.1 Introduction

In section 4.2, we present our classifier design. Our classifier ia ask to differentiate characters from non-characters, so now finding an algorithm that detects contours with a topologic structure to be characters will solve our problem. We decide to use an algorithm that computes the MSER of the source image. The algorithm was taken from D. Nister's and H. Stewenius' paper published in 2008 entitled "Linear Time *Maximally Stable Extremal Regions*" [14]. This algorithm claims to use less memory and has better cache-locality than other similar ones. We found the algorithm source code on OpenCV's GitHub repository. A result of this algorithm after **Non-Maximum suppression** was shown earlier in figure 18. After that we keep only the bounding boxes that have a strong probability to be characters using our algorithm named **ER Tracking**. Finally, we run our classifier that can recognize the characters from the remaining outliers. The pseudocode is shown below in figure 25.

An Extremal Region (**ER**), which is the output of the MSER algorithm, is a structure that contains multiple information for the OCR task. It is like a bounding box with extra information. This information is about:

1. MSER algorithm
    a. Seed point and threshold (max gray-level value)
2. Features of the bounding box
    a. Area
    b. Center
    c. Boundaries of the box (X, Y coordinates of the start and end point)
    d. The Color domains
3. OCR
    a. Classified character of the box
    b. Probability to be a letter

Figure 25. Pseudocode of Character detection unit.

```
1 Character_detection {
2      Detected_ERs = MSER(scr)//input: source image Out: detect the Ers
3
4       Non_max_ERs  = Non_Max_Suppressipn(Detected_ERs)//Decrease
5                           the number of Ers with Non-Maximum suppression
6      Tracked_Ers = Er_traking(Non_max_ERs )  // Track the Ers with
7                                strong possibility to be chars
8      Final_ERs = Classifier(Tracked_Ers)   // Run Classifier on DPU
9 }
```

3.3.2 MSER

The **linear MSER** algorithm first transforms the RGB image to YCrCb color space. MSER take as input the Y, Cr and Cb images along with the deference of their max and current intensity (255 - Y, 255 – Cr, 255 - Cb). These six inputs correspond to the different intensity values to detect an object.

In this paragraph we will now describe the algorithm from an abstract point of view as exactly mentioned in [14]. The algorithm needs the following data-structures:

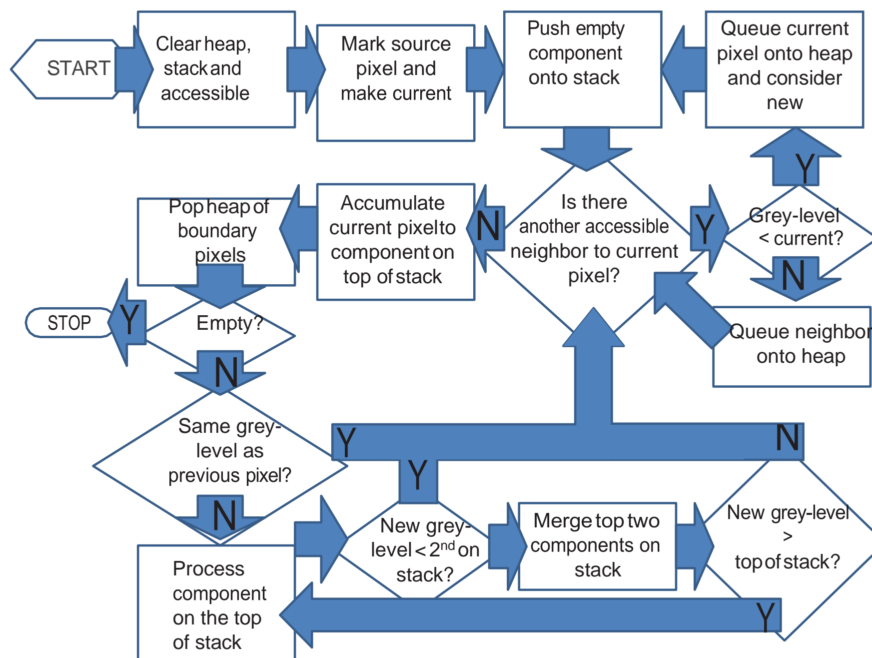


Figure 26: State graph for the algorithm [14].

- A binary mask of accessible pixels. These are the pixels to which the water already has access.
- A priority queue of boundary pixels, where priority is minus the grey-level. These pixels can be thought of as partially flooded pixels in the sense that water has access to the pixel in question, but has either not yet entered, or not yet explored all the edges out from the pixel. Along with the pixel id, an edge number indicating the next edge to be explored can be stored.
- A stack C of component information. Each entry holds the pixels in a component and/or the first and second order moments of the pixels in the component, as well as the size history of the component and the current grey-level at which the component is being processed. The maximum number of entries on the stack will be the number of grey-levels.

During execution of the algorithm for each of the six inputs, the components in the component info stack C may not correspond to components in the component tree. Rather, there will a number of components representing the 'down-stream' of water streaming

downhill towards a minimum, where each component is the set of pixels at a single grey-level that is part of the down-stream. A single component represents the pixels covered by the water currently filling back out of a minimum. The algorithm in a sense has two states, one where the down-stream is flowing downhill in search of a minimum, and one where a minimum has been found, and the water level is currently rising out of it.

To execute the algorithm, a pixel from which flooding will proceed is arbitrary chosen. This pixel can be thought of as the point at which water is being poured on, and the output result will be the same regardless of which pixel is selected, so we may simply pick the upper left corner of the image. We will refer to this as the source pixel. The algorithm is as follows, see also Figure 15:

1. Clear the accessible pixel mask, the heap of boundary pixels and the component stack. Push a dummy-component onto the stack, with grey-level higher than any allowed in the image.
2. Make the source pixel (with its first edge) the current pixel, mark it as accessible and store the grey-level of it in the variable current level.
3. Push an empty component with current level onto the component stack.
4. Explore the remaining edges to the neighbors of the current pixel, in order, as follows: For each neighbor, check if the neighbor is already accessible. If it is not, mark it as accessible and retrieve its grey-level. If the grey-level is not lower than the current one, push it onto the heap of boundary pixels. If on the other hand the grey-level is lower than the current one, enter the current pixel back into the queue of boundary pixels for later processing (with the next edge number), consider the new pixel and its grey-level and go to 3.
5. Accumulate the current pixel to the component at the top of the stack (water saturates the current pixel).
6. Pop the heap of boundary pixels. If the heap is empty, we are done. If the returned pixel is at the same grey-level as the previous, go to 4.
7. The returned pixel is at a higher grey-level, so we must now process all components on the component stack until we reach the higher grey-level. This is done with the **ProcessStack** sub-routine, see below. Then go to 4.

The ProcessStack sub-routine is as follows:

Sub-routine ProcessStack(new pixel grey level)

1. Process component on the top of the stack. The next grey-level is the minimum of new pixel grey level and the grey-level for the second component on the stack.
2. If new pixel grey level is smaller than the grey-level on the second component on the stack, set the top of stack grey-level to new pixel grey level and return from sub-routine (This occurs when the new pixel is at a grey-level for which there is not yet a component instantiated, so we let the top of stack be that level by just changing its grey-level.
3. Remove the top of stack and merge it into the second component on stack as follows: Add the first and second moment accumulators together and/or join the pixel lists. Either merge the histories of the components, or take the history from the winner. Note here that the top of stack should be considered one 'time-step' back, so its current size is part of the history. Therefore, the top of stack would be the winner if its current size is larger than the previous size of second on stack.
4. If(new pixel grey level>top of stack grey level) go to 1.

3.3.3 Non-Maximum suppression

One of the problems in linear MSER [14] is that the algorithm may find multiple detections of the same object. Remember that linear MSER will try to detect objects in six deferent input images, each of which correspond to a different intensity value. An object can be appearing more than once in these inputs, and rather than detecting an object just ones it might detect it more times. **Non-Maximum suppression** is a way to make sure that the algorithm detects this object only ones. An example of non-maximum suppression is shown in Figure 27.

The Non-Maximum Suppression algorithm is as follows:

1. Create a list of indexes by sorting them with the biggest Y end point of the bounding box.
2. Take the last index in the index list and add the index value to a list of picked indexes.
3. First find the largest (x, y) coordinates for the start of the bounding box and the smallest (x, y) coordinates of the end of the bounding box in the index list.
4. This coordinate can create a new bounding box, compute its Hight(H) and Width(W).

5. Compute the rotation vector of the overlap = (W*H)/area, where area is the area vector of the bounding boxes in the index list.
6. Delete all indexes from the index list that have overlap bigger than a threshold.
7. Keep looping until all indexes in the list are deleted.

The picked indexes that are related to the bounding boxes are the result.

Figure 27. Non-Maximum suppression example.

3.3.4 ER Tracking

Characters have a certain structure, they have common height, color, and topology. For example, characters that form a word are in the same line. That makes our problem a lot easier because we could scan the image and detect from the ERs, the bounding boxes that have this significant structure. This would help us decrease the number of outliers exponentially without a lot of computer resources. **ER Tracking** is not a sophisticated algorithm, with computational complexity of $O(\mathrm{NlogN})$ where N is the number of ERs after MSER algorithm. We can see the pseudocode in figure 28. This algorithm in order to understand that A, B ERs are in the same line, calculates in every iteration an adaptive threshold. This threshold must be less than the distance of A, B. This algorithm will extract

two large or two small bounding boxes that are alienated. Finally, we could run our classifier and detect all the characters.

Figure 28. ER Tracking pseudocode.

```
1 ER_Traking {
2      for(a in Ers) {
3          for(b in remaining Ers) {
4            if (a,b are in the same line) and (a,b have close colors) {
5                      Keep(a,b)//This means that they might be letters.
6                     }
7              }
8      }
9 }
```

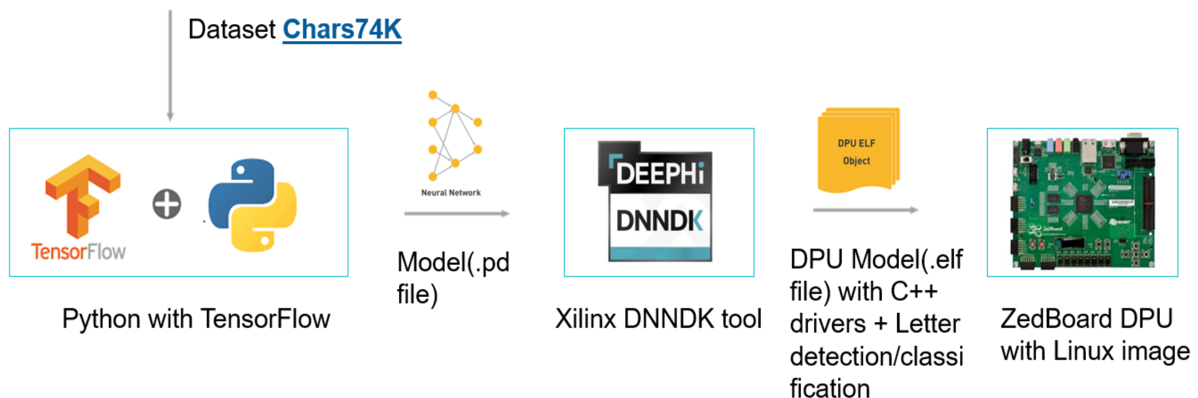
Chapter 5

ZOCR on ZedBoard Platform

5.1 NN Introduction

In this chapter we are going to see how we run our ZOCR application on ZedBoard FPGA. This OCR application consists of two separate tasks, the classifier and the character isolation unit. Our classifier is an NN model so it will use the DPU of ZedBoard. The character isolation unit is a software tool that will use the ARM CPU embedded on board. To quantile the DPU, NN must be transformed to a file identifiable by the bord (.elf). The final step is to combine the drivers with the characters Isolation/classification parts and use a Linux image to run OCR on ZedBoard. In figure 29, we illustrate a high-level project overview.

Figure 29. Project graph.



5.2 NN Model Deployment

5.2.1 NN Model Compression

In this subchapter we are going to see how we compress and prepare our model to import Xilinx tools. DNNDK tool is Xilinx's deep neural network development kit which can create the .elf file.

After training of the DCGAN, we save the discriminator model using the appropriate commands proposed by TensorFlow documentation [11]. In figure 30 we can see the files that create our model. The `data_set` folder contains all the character images from both the Char74K, and ICDAR 2003 and must be in this style for no errors. To run the train process, delete the `checkpoint` and `saved_model` folders and execute the `train.py` script. The

frozen_model_dnndk.pb is the frozen graph in which DNNDK tools will quantize the model [4]. It contains the weights and the biases along with the structure of the model. The *Create_DNNDK_files.py* take as input the *saves_model*, saved by TensorFlow and create the DNNDK frozen graph. In also prints the names of input and output nodes.

Figure 30. DCGAN folder.

```
$ sem-supervised_chars74k
        | -- data_set
        |     | -- img
        |          |-- natural_images_BadImag #\
        |          |-- natural_images_GoodImag# }Chars74k
        |     |-- ICDAR 2003
        | -- .py files
        | -- train.py
        | -- Create_DNNDK_files.py
        | -- chepoint
              | -- install.sh
              | -- samples
              | -- common
        | -- saved_model
              | -- saved_model.pb
        | -- frozen_model_dnndk.pb
```

The next step is to prepare the floating-point frozen model and dataset. The *data_gen.py* is shown as below in figure 31. This python script is used to preprocess and load the training images for our NN. As we can see, `Feature_Extraction()` class is responsible to preprocess and load our data.

Table 1: Input files for DECENT_Q [4].

No.	Name	description
1	**Frozen_graph**	**Located in** *${dnndk_chars74k}/frozen_model_dnndk.pb*
2	**Data_gen**	**A python function to read images in Chars74k dataset and do preprocess, locates in ${DNNDK_chars74k}/data_gen.py**

Figure 31: load_data file.

```
def load_data(iter):
    proj_directory="./"
    minibatch_size = 128
    real_size =(32,32,1)
    """
        This module loads the data and returns the sets to create the NN .
    """
    images = []
    image = Feature_Extraction(proj_directory = proj_directory,minibatch_size = minibatch_size,real_siz
    image.read_data()
    for index in range(0,minibatch_size):
        images.append(image.train_x[iter*minibatch_size+index])
    return {"input_real": images}
```

The third step is to run decent. Because we create a lot of files in ${*dnndk_chars74k}* directory we create a second one (${*dnndk_chars74k}*) where we include the appropriate scripts to create ".elf" file. DECENT takes a floating-point network model, pre-trained weights(float-32), and a calibration dataset as inputs to generate a lightweight quantized model with INT8 weights.

A script file in ${*dnndk_chars74k}* directory named *decent_q.sh* is shown as below in Figure 32. Run ssh decent_q.sh to invoke the DECENT_Q tool to perform quantization with the appropriate parameters. The inputs of the script file are the graph we saw later, the input and output nodes are the parameter names in the TensorFlow graph which are tensor names and are printed by *Create_DNNDK_files.py*. Input shape is the shape of the *input_real* and the *input_fn* is the function we discussed above. The method is by default the first which is proposed by Xilinx. The tool can also be installed using in the host machine with GPU support for faster results. The number of iterations is like the epochs and ten is best suited for this model.

Figure 32: decent_q script file.

```
decent_q quantize \
  --input_graph_def frozen_model_dnndk.pb \
  --input_nodes  input_real\
  --input_shapes ?,32,32,1 \
  --output_nodes discriminator/out \
  --input_fn data_den.load_data\
  --method 1 \
  --gpu 0 \
  --calib_iter 10 \
  --output_dir ./quantize_results \
```

The script may take several minutes to finish. Once quantization is done, the quantization summary will be displayed as below [4]:

```
INFO: Done Calibration
INFO: Start Generate Deploy Model
INFO: End Generate Deploy Model
********************* Quantization Summary *********************
INFO: Output:
  quantize_eval_model: ./quantize_results/quantize_eval_model.pb
  deploy_model: ./quantize_results/deploy_model.pb
```

Two files as shown in Table 1 will be generated under the ${*dnndk_chars74k}/quantize_results* directory. Then you could use the "*deploy_model.pb*" to compile the model using DNNC compiler and deploy it on DPU.

Table 1: DECENT_Q output files [4].

No.	Name	Description
1	**deploy_model.pb**	**Quantized model for DNNC (extended Tensorflow format)**
2	**quantize_eval_model.pb**	**Quantized model for evaluation and dump**

Finally, we evaluate the quantized model. One python file named "*eval.py*" can be found in *${dnndk_chars74k},* it is used to perform evaluation for the float and quantized model, respectively. The validation dataset is stored in the *${semi supervised_chars74k}/ Dataset/ICDCAR 2003*. This dataset contains the validation and test images from RCR.

Run the `evaluate_dnndk_model()` located in the `eval` python script file, to perform evaluation. We must specify the input and output nodes from the frozen graph. For this example, the accuracy on this data should be 78% to continue the process. Once evaluation is done, the accuracy of the model will be displayed as below:

Figure 33: Evaluation results of the float and quantizes model.

```
from eval import evaluate_dnndk_model
evaluate_dnndk_model(proj_directory = proj_directory,input_node = "input_real",output_node = "discriminator/out")

11492 data successfully loades with shape :  (32, 32, 1)

Classifier test accuracy 0.78438905
```

5.2.2 NN Model Compilation

DNNC can support TensorFlow model compilation into an executable file undeniable by the DPU. DNNDK provides *dnnc.sh* script files that with some adjustments can compile our model specifically for ZedBoard FPGA.

The script file dnnc_Zedboar.sh under ${*dnndk _chars74k*}, for compiling TensorFlow model, is shown in the following figure. For TensorFlow model compilation, you must specify the parser type using TensorFlow through the -- parser option, otherwise DNNC will display an error.

```
net="chars74k"
CPU_ARCH="arm64"
DNNC_MODE="debug"
dnndk_board="ZedBoard"
dnndk_dcf="../../../dcf/ZedBoard.dcf"

echo "Compiling Network ${net}"

# Work space directory
work_dir=$(pwd)

# Path of caffe quantization model
model_dir=${work_dir}/quantize_results
# Output directory
output_dir="dnnc_output"

tf_model=${model_dir}/deploy_model.pb
```

Figure 34: DNNC Compilation Script for TensorFlow chars74k

In Figure 35 we can see DNNC output information when compilation is completed successfully. DNNC compiles the NN model into an equivalent DPU assembly file, which is then assembled into one ELF object file. For this application, our NN model was named after the dataset to chars74k. This also shows the information about layers unsupported by the DPU. The DPU ELF object file is regarded as DPU kernel, which then becomes one execution unit from the perspective of runtime N2Cube after invoking the API *dpuLoadKernel()*. N2Cube loads the DPU kernel, including the DPU instructions and network parameters, into the DPU dedicated memory space and allocate hardware resources. After that, each DPU kernel can be instantiated into several DPU tasks by calling *dpuCreateTask()* to enable the

multithreaded programming. The network model is compiled and transformed into two different kernels:

- Kernel 0: chars74k_0 (Run on DPU)
- Kernel 1: chars74k _1 (Deploy on CPU)

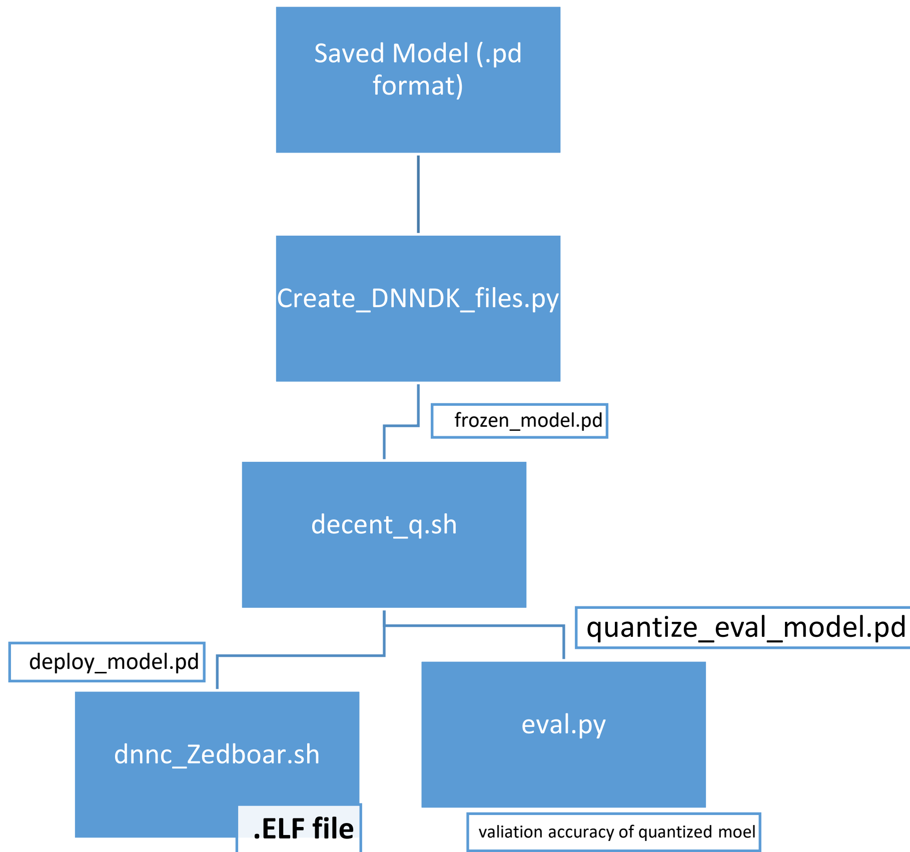
The kernel _0 runs on the DPU. DNNC generates an ELF object file for this kernel in the `output_dir` directory. The other kernel is for "SoftMax" operation, which is not supported by DPU and must be deployed and run on the CPU. Workload **MACs** reference to the total computation workload of the NN in the PE unit, without including the computation of bias. In other words, chars74k NN executes 14.47 million operations per classification.

```
DNNC Kernel topology "chars74k_kernel_graph.jpg" for network "chars74k"
DNNC kernel list info for network "chars74k"
                               Kernel ID : Name
                                       0 : chars74k_0
                                       1 : chars74k_1

                             Kernel Name : chars74k_0
--------------------------------------------------------------------------------
                             Kernel Type : DPUKernel
                               Code Size : 0.01MB
                              Param Size : 0.43MB
                           Workload MACs : 14.47MOPS
                         IO Memory Space : 0.02MB
                              Mean Value : 0, 0, 0,
                              Node Count : 7
                            Tensor Count : 8
                    Input Node(s)(H*W*C)
          discriminator_conv2d_Conv2D(0) : 32*32*1
                   Output Node(s)(H*W*C)
           discriminator_dense_MatMul(0) : 1*1*63
```

Figure 35: DNNC Compilation Log for TensorFlow chras74k

The following graph illustrates the steps for creating the .elf file as a summary for virtualization purposes:



5.3 Run ZOCR on ZedBoard DPU

Connect the ZedBoard to a LAN using ethernet. With an ethernet connection established, you can copy the DNNDK package from the host machine to the evaluation board. The steps below illustrate how to setup DNNDK running environment for ZedBoard, provided that DNNDK package is stored on a Linux system.

Boot ZedBoard by opening a terminal and run minicom. Find the IP of the board with ifconfig. Copy the file *${OCR_ZedBoard}* on the board. In this case, use the following commands to extract and copy the package with IP address 192.168.0.10 of the board:

`scp -r xilinx_dnndk_v3.1/ZedBoard root@192.168.0.10:~/` (default password is **root**)

`scp -r ./OCR_ZedBoard   root@192.168.0.10:~/`

1. Log in to ZedBoard board, change to the *${ZedBoard}* directory and run install.sh

`ssh root@192.168.0.10:~/` #Log in to ZedBoard

```
~/OCR_ZedBoard $./install.sh # to install DNNDK tools to Zedborad
```

The messages "Complete installation successfully" will display if the installation completes successfully.

2. Use the make file to build OCR code and execute the binaries.

```
~/OCR_ZedBoard$make
~/OCR_ZedBoard$./OCR -i img.png
```

After the execution of the code the predicted words will appear in the terminal. The application will also save a result image under the *${OCR_ZedBoard/result.png}* for virtualization purposes. We can see a result image in figure 36.

Figure 36: ZOCR result image.

Chapter 6

Experimental Results

The proposed OCR application was evaluated in the ICDAR 2003 dataset. As it was mentioned in a previous chapter, ICDAR 2003 dataset contains the RRTL and RCR sub-dataset. In figure 37 we can see how RRTL and RCR are associated. The RRTL (37a) contains natural scene images, while RCR (37b) contains natural character images extracted from the RCR. We want to validate the accuracy of our NN with the RCR, and the accuracy of character isolation-localization with the RRTL. We also provide a comparison between some of the most know OCR applications.

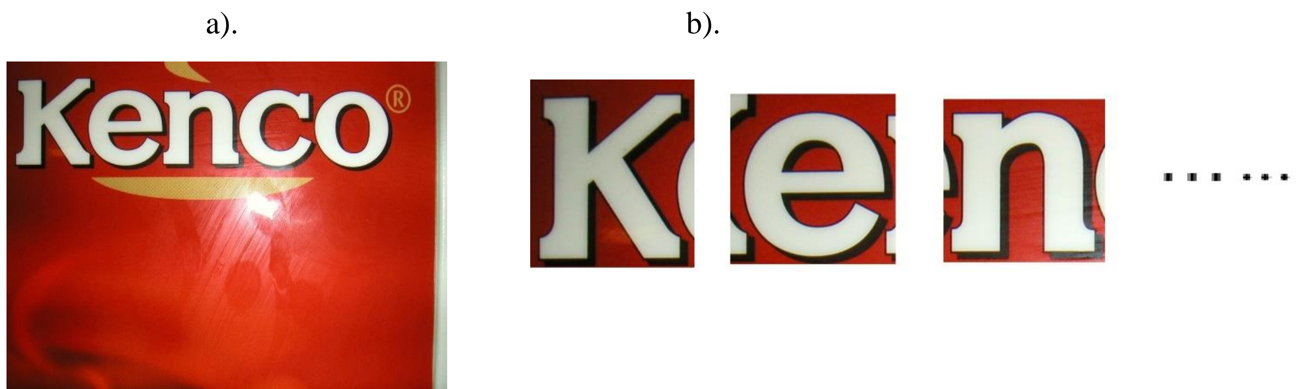


Figure 37: RRTL vs RCR.

6.1 NN Performance

To calculate the accuracy of our NN, we download the RCR sub-dataset which is like the chars74K dataset. We split this dataset into a test and validation set, to calculate the accuracy of the model in deferent distributed samples. The validation set is used to calculate the accuracy during training, and the test set is used to calculate the accuracy after training is finished. The accuracy of the model is **78%** where 77% was the test set and 80% the validation set. Figure 38 presents the validation accuracy (Y-axis) of the model during training for each second. We train it for three epochs. This hyperparameter determines the number of times the model learn from the entire dataset. We noted that if we train it with more epochs the accuracy will increased slightly, but then the model could not recognize characters from non-characters (real-vs-fake class).

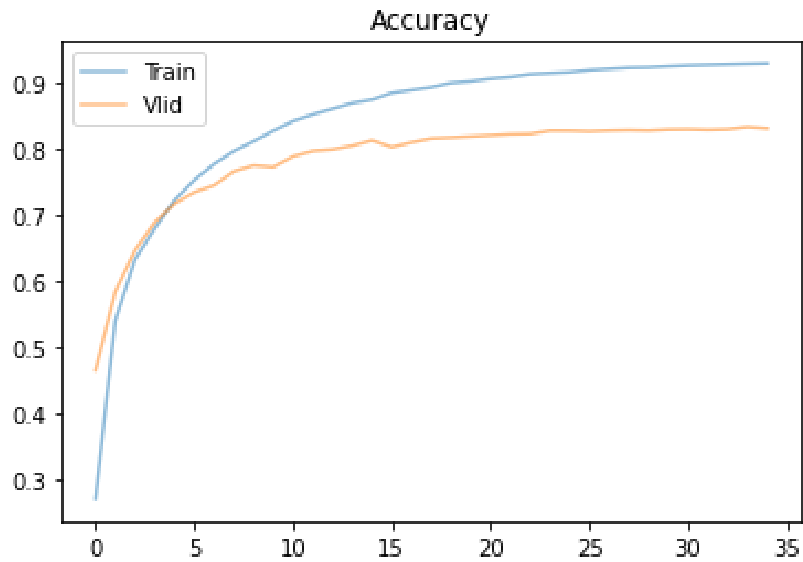


Figure 38. Model train and validation accuracy.

After a failure analysis, we noticed that 2% was from pure error. Also, 6% was from non-distributed characters like the example in figure 39a. We can see that this “R” does not look like the “R” images from the dataset. A rate of 3% was from some images of vary bad quality like the example in figure 39b. Finally, 11% of the error was some confusion that can be understand, due to 11 duplicate character classes like:

- C with c
- I with l
- K with k
- 0 with o and O
- P with p
- S with s
- etc.

b). a).

Figure 39. Miss-classified images.

It is noticeable that, we can increase accuracy of ZOCR by making a small misspelling correction to these 11 duplicate character classes. When a character of these duplicate classes is detected, we could easily change its upper-case or lower-case depending on the previous character of the word. We call this technique the **upper/low-case check**. This will increase our ZOCR recognition accuracy to **89%**.

We also test the real-vs-fake performance in the RCR sub-dataset. During the test of the NN we calculate for each image in the dataset the probability to be character (**P(real-vs-fake)**). We noticed that **92%** (called **P(identification)**) of this probability values were greater than 95%. We then took a picture from the street and run our NN to see its performance with another distributed image. In figure 40 we can see the results from this image. In the left image we run the OCR application and we print for each bounding box the detected class and the probability ***P*** to be a character. We can see that for the characters the *P* value is 95% or greater and the outliers (in the bottom corner) have *P* value significantly lower. In the right image of figure 40, we can see the OCR application if we apply a threshold of 95% to these *P* values. Most of the outliers were extracted from the image except from the exclamation mark that mistakenly identified as a character.

6.1.1 NN Performance Comparison

To compare our NN with the most known stare-of-the-art methods, we calculate the performance of our NN in word recognition. To calculate the accuracy of the NN in word recognition, we randomly choose 50 words from the ICDAR 2003 dataset, and we truck each individual character in the RCR sub-dataset. From these characters we found the misclassified ones related to some words from the ICDAR 2003 dataset i.e., these words are misclassified. The accuracy that was measured was **84,64%**, using the upper/low-case check technique. In table 2, we can see the performance comparison with some of the most known word recognition methods. In the IC03, which is the dataset, we can see the Scene text recognition accuracies (%) for each method. In the second row, “50”, and “Full” denote the lexicon used, and “None” denotes recognition without a lexicon. Table also presents, the classification technique used in each method. SVM is the support vector machine, a model that can map the input data to a higher dimensional space to assign a decision boundary. Parameters (**Params.**) correspond to the number of weights and biases of the NN. We also provide the floating-point operations (**FLOPs**) or linear operation for a few of

the methods. To calculate our model's FLOPs, we multiple "10" times the MACs. "10" is the average number of characters in a word of IC03 dataset, and MACs, as we said in the previous chapter, represent the number of multiplications and accumulations in the PE unit calculated by DNNC. Thus, our system executes 144 million linear operations on average to predict a word, which is quite lower than the other methods. In conclusion, we implement a recognition model which is more efficient than other state-of-the-art methods, due to its vary low computational complexity and its high enough accuracy.

Figure 40. Run OCR application.

Methods	IC03			Classifier	Param.	FLOPs
	50	Full	0		F-32	M-A
Wang *et al.* [55]	76.0	62.0	-	SVM	-	-
Wang *et al.* [56]	90.0	84.0	-	CNN+SVM	-	-
Bissacco *et al.* [57]	90.4	78.0	-	DNN	-	-
Jaderberg *et al.* [52]	96.2	91.5	-	DCNN	300M	-
Su and Lu [58]	92.0	82.0	-	DCRNN	-	-
Jaderberg *et al.* [47]	98.7	**98.6**	93.1	DCNN	500M	-
Jaderberg *et al.* [48]	97.8	97.0	89.6	DCNN	304M	-
Shi *et al.* [49]	98.7	98.0	91.9	DCRNN	8.3 M	≈10.9B
Lee *et al.* [50]	97.9	97.0	88.7	DCRNN	2.9M	2B
Yang *et al.* [51]	97.7	-	-	DCRNN	144M	-
Cheng *et al.* [53]	99.2	97.3	94.2	DCRNN	-	-
Cheng *et al.* [59]	98.5	97.1	91.5	DCRNN	-	-
W. Liu *et al.* [54]	96.9	95.3	89.9	DCRNN	48.7M	-
SAR (Li et all.) [45]	98.8	-	-	DCRNN	44M	16.4B
SATRN (Lee *et al.)* [46]	96.7	-	**-**	DCRNN	55M	35.9B
Shi *et al.* [60]	**98.8**	98.0	94.5	DCRNN	-	-
ZOCR **(ours)**	-	-	84.6	DCNN	**144K**	**144M**

Table 2: Performance results of the most known word recognition methods.

6.1.2 DPU Performance Comparison

To compere the performance of Xilinx ZedBoard DPU, we calculate the time taken needed to run our NN for a singe classification in three deferent systems. The systems used are:

- CPU: Intel(R) Core(TM) i5-7200U CPU 2.50GHz, 2 Cores
- GPU: Nvidia 1060 3GB
- Xilinx ZedBoard DPU

For our calculations to be as precise as possible, we run for each system our NN one hundred times and calculate the average time. Table 3 presents these calculations. We can see that DPU using the quantized model with 8-bit integers can run approximately 45% faster than a GPU and 150% faster than a CPU.

CPU (F-32)	GPU (F-32)	DPU (INT8)
1317 us	769 us	540 us

Table 3: Time taken needed for a single classification in three deferent systems.

6.2 Characters Isolation Performance

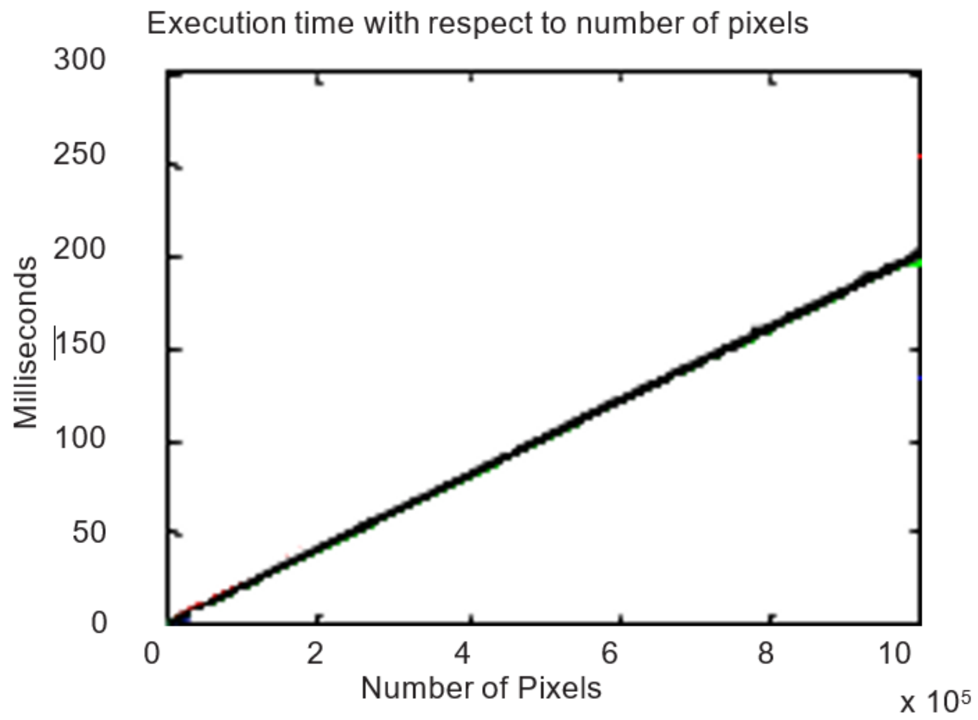
Character isolation task first runs the linear MSER to extract potential characters. The accuracy of linear MSER in the RRTL sub-dataset was **70%**. After a failure analysis, we noticed that the detection works without any error when the characters of the given image defer from the background.

According to the paper, the time required for linear MSER, can be shown to be bounded by $O(\mathrm{N}\alpha(N))$, where N is the number of pixels and $\alpha(N)$ is the inverse of the Ackermann function, whose value is smaller than 5 if N is of the order 10^8. Figure 9 presents the execution time of linear MSER as a function of image size on square images ranging from one pixel to one mega-pixel. The pixels are from a natural scene image like the one in ICDAR dataset. This experiment ran with single-threaded code on a laptop with 3GB of RAM and an Intel T7400 2.16GHz processor. We also noticed that if we resize the input image of the dataset to a fixed size of 600x480, the accuracy will not be decreased.

After MSER (and ER extraction and Non-Max suppression) we run our NN to localize the characters and extract the outlier. Our NN can identify without error the characters of IC03 dataset. This is because, NN cannot identify only the 8% of the IC03 characters (1-P(identification)), and these characters weren't tracked by linear MSER. Which means that, the probability of character localization in the IC03 is 70%. This is the precision of character localization. We also noticed that 4% of the outliers was mistaken recognized as character from the NN. After a failure analysis we nosed that the error was from some symbols that looked close enough with characters e.g., question mark ("?"), exclamation mark ("!") etc. The demanding time for this step is $\boldsymbol{Outliers * DPU_time}$, where Outliers is the number of outliers after implementing ER tracking algorithm and DPU_time is the time of DPU to run the NN. To calculate the execution time of the NN in the DPU, we run 10-character images and we measure the mean of their execution time. This time is **540 μs.** This OCR application can detect and classify letters even if the image is rotated. The classification of

each detected character can be done in between (-30, +30) degrees. The detection part can be accomplished no matter the degrees.

Figure 41. Execution time of linear MSER [15].



Chapter 7

Conclusions and future work

OCR systems are very important in our world and can make our lives a lot easier. In professional sector they can achieve high productivity, cost reduction, and provide high accuracy and accessibility. For example, a business that receives a lot of documents daily and needs to search them, now in real time they can transform the paper documents into digital ones and track a key word without going through the extensive slot of files. Another application of OCRs is that it can help people with learning difficulties such as dyslexia. People with dyslexia have the ability understand a problem (e.g., mathematical problem) by listening rather that by reading it. So, we could transform this problem to a digital representation and then with another tool to a speech representation. Once the problem is converted to speech people with dyslexia can hear and understand the problem in shorter time.

Our OCR system is developed to run on a standard local hardware and can be trained on a custom dataset for specific applications. It computes 90% less linear operations per image and the time taken is significantly lower than others.

In the future we plan to increase the efficiency and the performance of our application. We also want to run it on cloud instance to solve real IoT problems that need a faster OCR. At first, we want to take as input frame data coming from a camera connected to the board. Moreover, we want to perform the OCR task to Greek words. To achieve this, we need to train our model with 27 extra Greek character classes using our own dataset. That would be challenging because creating our own dataset need to collect eight thousand characters from natural images and label them. We also want to implement the most used functions of our model like SoftMax on Hardware-LUTs for even faster results. Finally, by using another detecting technique we want to increase the accuracy of the detection unit. In the paper with title “Detecting Text in Natural Scenes with Stroke Width Transform” [7] is proposed a new algorithm that can detect characters with even better accuracy than linear MSER.

REFRENCES

[1] S. M. Lucas, A. Panaretos, L. Sosa, A. Tang, S. Wong, R. Young (2003): "ICDAR 2003 Robust Reading Competitions", Proc. Seventh Int'l. Conf. on Document Analysis and Recognition (ICDAR).

[2] Alec Radford, Luke Metz*, and* Soumith Chintala (2015): "Unsupervised representation learning with deep convolutional generative adversarial networks".

[3] Tim Salimans, Ian J. Goodfellow, Wojciech Zaremba, Vicki Cheung, Alec Radford, Xi Chen (2016): "Improved Techniques for Training GANs"

[4] Xilinx. DNNDK User Guide (UG1327) v.16 2019,

https://www.xilinx.com/support/documentation/sw_manuals/ai_inference/v1_6/ug1327-dnndk-user-guide.pdf

[5] OpenCV Scene Text Detection and Recognition in Natural Scene Images, https://github.com/opencv/opencv_contrib/tree/master/modules/text

[6] H. Cho, M. Sung and B. Jun, "Canny Text Detector: Fast and robust scene text localization algorithm", Computer Vision and Pattern Recognition, 2016.

[7] B. Epshtein, E. Ofek, and Y. Wexler, "Detecting text in natural scenes with stroke width transform", Computer Vision and Pattern Recognition

[8] [online] Stroke Width Transform with python, https://github.com/sunsided/stroke-width-transform

[9] [online] Available: Chars74K dataset, http://www.ee.surrey.ac.uklCVSSP/demos/chars74k.

[10] Schalkoff, R.J., 1997. Artificial Neural Networks. McGraw-Hill, "Demystifying knowledge acquiring black boxes". IEEE Trans. New York.

[11] [online] TensorFlow 1.12 documentation, https://www.tensorflow.org/versions/r1.15/api_docs/python/tf

[12] [online] OpenCV documentation, https://docs.opencv.org/master/index.html

[13] Raymond Smith, Chunhui Gu, Dar-Shyang Lee, Huiyi Hu, Ranjith Unnikrishnan, Julian Ibarz, Sacha Arnoud, Sophia Lin (2016): End-to-End Interpretation of the French Street Name Signs Dataset.

[14] [online] Artificial Neural Networks Wikipedia https://en.wikipedia.org/wiki/Artificial_neural_network

[15] Nistér D., Stewénius H. (2008): Linear Time Maximally Stable Extremal https://doi.org/10.1007/978-3-540-88688-4_14

[16] [online] OpenCV MSER Class References, https://docs.opencv.org/master/d3/d28/classcv_1_1MSER.html

[17] [online] OpenAI official site, https://openai.com/

[18] Schantz, H. F. (1982): The history of OCR: optical character recognition, Recognition Technologies Users Association.

[19] D'Albe, E. E. Fournier (1914): “On a Type-Reading Optophone”, https://royalsocietypublishing.org/doi/pdf/10.1098/rspa.1914.0061

[20] Gerardus Blokdyk (2018): “Recurrent neural network: Real Life Actions Paperback “

[21] [online] Xilinx ZedBoard, xilinx.com/support/university/boards-portfolio/xup-boards/DigilentZedBoard.html

[22] Sheikh Faisal Rashid (2014): Optical Character Recognition - A Combined ANN/HMM Approach

[23] Rosenblatt F. (1958): "The Perceptron: A Probabilistic Model For Information Storage And Organization in the Brain"

[24] Schmidhuber, J. (2015): "Deep Learning in Neural Networks: An Overview". Neural Networks.

[25] O. Russakovsky, Hao Su, ..., Li Fei-Fei (2010): ImageNet Large Scale Visual Recognition Challenge

[26] Ashwin Vijayakumar (2019): A Machine Learning Approach for Document Image Repair and Enhancement of Severely Degraded Printed Text

[27] Sepp Hochreiter and Jürgen Schmidhuber (1997): Long Short-Term memory. Neural computation

[28] [online] YCrCb color spaces, learnopencv.com/color-spaces-in-opencv-cpp-python/

[29] [online] Sobel Filter, docs.opencv.org/3.4/d2/d2c/tutorial_sobel_derivatives.html

[30] Irwin Sobel (2014): History and Definition of the Sobel Operator

[31] G. E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov (2012): Improving Neural Networks by Preventing Co-adaptation of Feature Detectors

[32] F. Schilling (2016): The Effect of Batch Normalization on Deep Convolutional Neural Networks

[33] Diederik Kingma, Jimmy Ba (2015): "Adam: A Method for Stochastic Optimization"

[34] Yujia Li, Kevin Swersky, and Richard S. Zemel. (2015): Generative moment matching networks.

[35] [online] VGG Net.: https://neurohive.io/en/popular-networks/vgg16/

[36] N. Srivastava, ..., R. Salakhutdinov (2015): Dropout: A Simple Way to Prevent Neural Networks from Overfitting

[37] Sergey Ioffe, Christian Szegedy (2015): Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift

[38] S. J. Nowlan, J. C. Platt (1994): A convolutional neural network hand tracker, in: Proceedings of the Advances in Neural Information Processing Systems (NIPS)

[39] [online] FSNS dataset: https://rrc.cvc.uab.es/?ch=6

[40] Valueva, M.V.; Nagornov, N.N.; Lyakhov, P.A.; Valuev, G.V.; Chervyakov, N.I. (2020): Application of the residue number system to reduce hardware costs of the convolutional neural network implementation

[41] Yanghua Jin, ..., Zhihao Fang (2017): Towards the Automatic Anime Characters Creation with Generative Adversarial Networks

[42] [Online] Hardware Description Languages: paginas.fe.up.pt/~jcf/ensino/disciplinas/mieec/pcvlsi/2010-11/cmosvlsidesign_4e_App.pdf

[43] [Online] Xilinx DPU: https://www.xilinx.com/support/documentation/ip_documentation/dpu/v3_0/pg338-dpu.pdf

[44] [Online] Python: https://www.python.org/

[45] Li, H.; Wang, P.; Shen, C.; and Zhang, G. 2019. Show, attend and read: A simple and strong baseline for irregular text recognition. In The Thirty-Third AAAI Conference on Artificial Intelligence

[46] Junyeop Lee, Sungrae Park, et al (2019): On Recognizing Texts of Arbitrary Shapes with 2D Self-Attention

[47] M. Jaderberg, K. Simonyan, A. Vedaldi, and A. Zisserman (2015): Reading text in the wild with convolutional neural networks

[48] M. Jaderberg, K. Simonyan, A. Vedaldi, and A. Zisserman (2015): Deep structured output learning for unconstrained text recognition.

[49] Shi, X. Bai, and C. Yao (2017): An end-to-end trainable neural network for image-based sequence recognition and its application to scene text recognition.

[50] C.-Y. Lee and S. Osindero (2016) Recursive recurrent nets with attention modeling for ocr in the wild.

[51] X. Yang, D. He, Z. Zhou, D. Kifer, and C. L. Giles. (2017): Learning to read irregular text with attention mechanisms.

[52] M. Jaderberg, A. Vedaldi, and A. Zisserman. (2014): Deep features for text spotting.

[53] Z. Cheng, F. Bai, Y. Xu, G. Zheng, S. Pu, and S. Zhou (2017): Focusing attention: Towards accurate text recognition in natural images.

[54] Wei Liu, Chaofeng Chen, Kwan-Yee K Wong, Zhizhong Su, and Junyu Han (2016): Star-net: A spatial attention residue network for scene text recognition.

[55] K. Wang, B. Babenko, and S. J. Belongie (2011): End-to-end scene text recognition.

[56] T.Wang, D. J.Wu, A. Coates, and A. Y. Ng. (2012): End-to-end text recognition with convolutional neural networks.

[57] A. Bissacco, M. Cummins, Y. Netzer, and H. Neven. (2013): Photoocr: Reading text in uncontrolled conditions

[58] B. Su and S. Lu. (2014): Accurate scene text recognition based on recurrent neural network.

[59] Z. Cheng, Y. Xu, F. Bai, Y. Niu, S. Pu, and S. Zhou. (2019): Aon: Towards arbitrarily-oriented text recognition.

[60] B. Shi, M. Yang, X.Wang, P. Lyu, C. Yao, and X. Bai. (2018): Aster: an attentional scene text recognizer with flexible rectification.