Automatic parsing of hand-written Turing machines for assisted evaluation

Johannes Mario Meissner Blanco

August 6, 2019

Contents

Resumen

Este trabajo revisa los conceptos básicos de redes neuronales, al mismo tiempo que estudia el más avanzado y emergente campo del deep learning. En base a estos conocimientos, se diseña y entrena un modelo de red neuronal capaz de realizar reconocimiento óptico secuencial de caracteres sobre definiciones de Máquinas Turing manuscritas para su evaluación automatizada en exámenes. Una implementación completa e independiente es proporcionada en forma de repositorio de código abierto. Dicho software permite modificar y ampliar el set de datos de entrenamiento existente para mejorar el modelo. Detalles sobre el funcionamiento del sistema también se detallan en este documento.

Keywords: *máquina de Turing, deep learning, red neuronal, evaluación automatizada*

Abstract

This project reviews the basic concepts of neural networks, as well as studies the more advanced and emerging field of deep learning. This knowledge is then applied to design and train a machine learning model capable of performing sequential optical character recognition on handwritten Turing machine definitions with the purpose of automating their evaluation in exams. A fully independent and complete implementation is provided in a freely available software repository[1]. The software allows to modify or amplify the existing training set as to keep improving the current model. Details on how the system works are also provided in the current document.

Keywords: *Turing machine, deep learning, neural network, automatic evaluation*

[1]`https://github.com/mariomeissner/deep_turing_ocr`

1 Introduction

The motivation for this project is manifold. First and foremost, its goal is to gather and formalize the basic concepts of deep neural networks (a field also sometimes called deep learning), and apply them to a real world problem. The extensive theoretical research and concept solidification that has been carried out can be appreciated in Sections 1.2 through 3. The field is currently one of the most promising paths forward in the advancement of AI technology. Growing really quickly and not showing any signs of stopping any time soon, it can be an invaluable asset in one's career.

Secondly, there is a large knowledge gap between theoretical machine learning and the actual implementation of a stand-alone system capable of offering a pleasing user experience. Of course, theoretical knowledge is essential to build the model, and is important in order to obtain any valuable prediction. However, the set of skills needed to actually write the code to build and train a neural network is, surprisingly, not as overlapping as it may seem. Moreover, embedding the trained model into a production system is yet again very different. Experiencing the whole process is certainly a necessity for anyone willing to work in this field of the industry.

Lastly, the automation of repetitive tasks has personally always been a hobby for me. The evaluation procedure for a specific course of the Computer Science degree at the University of Cantabria had recently caught my eye. One of the most tedious segments of this evaluation is the Turing machine, composed of tens or even hundreds of lines defining transitions between states. In order to check if the Turing machine is performing the desired task correctly, the evaluator has to copy all lines into a text file on the computer, and then run a simulation script to evaluate it. This problem immediately sparked interest, and it would be a perfect candidate to apply deep learning models to speed up the process.

Evaluating student's knowledge on paper-based exams instead of computer-based is preferred for many professors due to the fact that it allows for better capturing of the knowledge the student has been able to gain. As an example, a mathematics exam for elementary school may involve many small calculation steps, which could be trivially solved if using a computer or calculator. However, the goal is that students learn to mentally perform these calculations. In the case of writing Turing machines, using a computer could allow the student to evaluate the machine he or she has written, and obtain feedback on the mistakes that it contains. Nevertheless, by evaluating these machines by hand and simulating its execution without external tools, students prove that they have acquired a high level understanding of this computational model.

This project can be a good resource for any professional looking into automating tedious handwriting-to-text tasks, with only small modifications needed to make it work in a wide variety of situations.

The next section presents some current approaches by citing the most relevant work on the topic.

1.1 Research and Available Approaches

The aim of the project, automating the evaluation of student-generated content, is not new. As an example, in [5] the authors describe an automatic essay scoring system using text mining techniques. Similarity between new text and a set of hand-labelled texts is calculated to assess its quality. In many cases, the system agreed with the judgement of experts. In [2], they study the correlation between the correct usage of natural language and grades in mathematics courses, opening interesting paths toward assisted grading tools. In order to obtain a textual representation of student's exams, as is required in the present project, two key steps are necessary: finding the relevant segments in the image, and converting them into text using computer vision techniques.

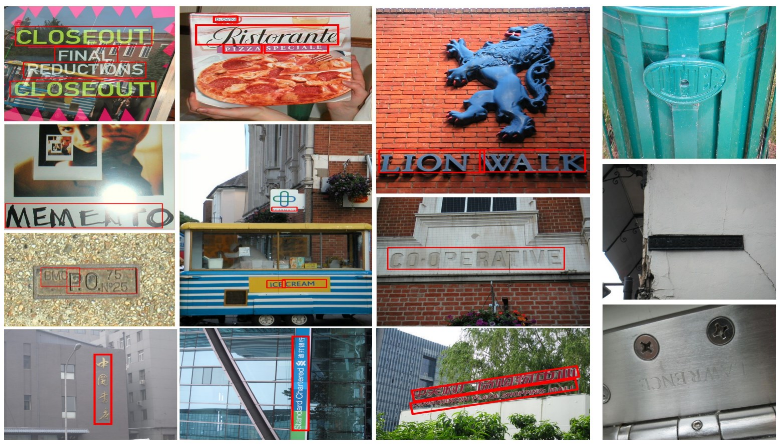


Figure 1: Taken from [24]. A set of scene images containing text. The red boxes are one of the tasks that the model can be trained to perform: finding the bounding boxes of the text.

The task of finding and recognizing text in images (called *Image-based Sequence Recognition*) is as old as computer vision itself, but is still an active research area that has recently experienced important advancements thanks to deep learning. Some important publications include *Text-CNN* (Text Convolutional Neural Network) [8] and *CRNN* (Convolutional Recurrent Neural Network) [24]. Both are deep learning based machine learning models that locate the regions in the image where text can be found, and then produce a textual output.

An example set of candidate images can bee seen in Figure 1, where bounding box prediction is performed. Recent papers such as *PixelLink* [3] also show that there are still many alternative ways to carry out this task. They utilize a pixel-based approach for region segmentation. A more applied example that is inspired by PixelLink can be seen in [11], where the researchers work on performing text detection on whiteboards, a task very related to what will be done in this thesis.

Bounding box prediction (the first of the two steps required), also called region proposal, is important not only in text recognition but also for computer vision in general. Commonly, this task is coupled with image classification, such that the model is capable of predicting what class each box belongs to. A landmark technique in this area is the family of *R-CNN* (Region Convolutional Neural Network) architectures [6, 19], which have a component called a *Region Proposal Network* (RPN) to suggest which segments of the image are worth looking at, and then a second network is applied on those selected segments to perform feature extraction and classification. A review of RPNs can be found in [10]. Other techniques capable of performing this task include the "You Only Look Once" architecture [18], which claims to be much faster by only processing ("looking at") the image once, and not several times like for example R-CNNs do.

A diagram representing the architecture of a CRNN can be found in Figure 2. Given an image with text, such as the input image displayed at the bottom, the model will output the word contained, in this case "state".

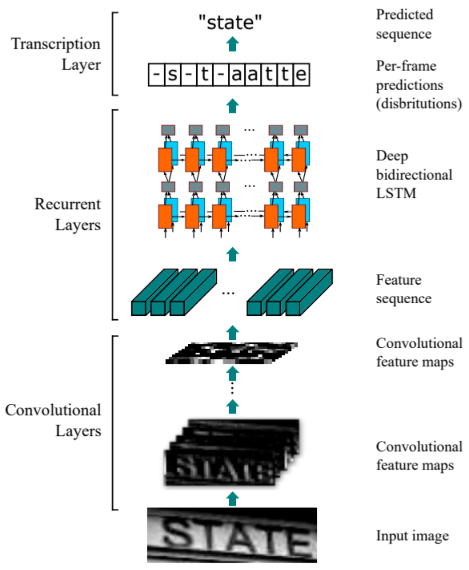


Figure 2: The complete structure of the CRNN network [8].

The Turing Machine Recognition project requires to find a very high number of lines of text, followed by an OCR (Optical Character Recognition) step for each of those lines. The two models mentioned in the first paragraph are not meant to be used for so many different text instances in the same image. The PixelLink architectures are built to be able to deal with texts with big size or rotation differences, and their complexity is unnecessarily high for what is trying to be accomplished here. Thus, the problem was approached by separating the two steps. On one hand, tackling the issue of finding the box coordinates of each line, and on the other, performing OCR on these boxes, each containing only one line.

The first step could be done with a technique such as "You Only Look Once", but would involve tuning the model for this specific use-case, and building a dedicated dataset. This can easily become a full project on its own. An alternative is to use existing tools such as the *Tesseract OCR* library. Although the accuracy for this specific task is not as high as one would desire, it is a good first approach.

The second step is what this project is focused on. Following a CRNN structure [24], a model was built and trained to perform OCR on the boxed Turing definition lines, as will be thoroughly explained in the different sections of this thesis.

1.2 The History of Neural Networks

Neural networks represent the core of this project, and reviewing their history is an important step to better understand their current popularity.

Neural networks have been present in the field of machine learning since the mid 20th century, when the perceptron algorithm started to be used as a linear classifier to produce binary classifications [20]. The early perceptron algorithm, and its improvement, the pocket algorithm, iterate over the dataset trying to find the best hyperplane to separate two clusters of labeled data. The internal parameters of the perceptron, representative of this hyperplane, are updated every time a data point is on the wrong side of it by modifying the values in a way that puts the point back to the correct side, disregarding however the position of all other points. The hope is that over time, the hyperplane finds a position that is satisfactory for most points. One would then need to manually tell the perceptron to stop (or program it to run for a specific amount of iterations), and the current set of parameters will be the result. The biggest problem with this algorithm was that, many times, an adjustment made to the hyperplane to correct one data point would actually worsen the position of it overall (especially if there are outsiders). To solve this, the pocket algorithm will "pocket" the best hyperplane it has found by comparing the overall accuracy of the parameters at each iteration with the saved ones. When stopping the loop, instead of returning the last found hyperplane, it can now return the pocketed one, which is usually better.

If we connect many perceptrons in a layer, and work on some other modifications like the learning algorithm used to adjust the parameters, we can achieve much more precise results and fit data in many different ways, not only in a linear fashion. This ultimately evolved into what we know as neural networks. However, the field stagnated due to the limitations of the simpler models and the high computational power required to run the more complex ones. Specifically, as Werbos, P.J. detailed in his paper [27], simple behavior like the XOR gate cannot be reproduced with perceptrons. Minsky et al. [15] also mention this problem and additionally acknowledge the complexity and lack of feasibility of running the bigger, more capable networks.

As later in the century more complex models with several layers emerged [22], it was possible to classify data with more precision. Highly non-linear functions can be applied to correctly group data into different categories. However, their popularity remained marginal. Other models like Support Vector Machines were crushing the machine learning landscape with better performance, higher accuracy and less computational power required. These models are also mathematically provable and their inner workings are clear and easily interpretable. It is possible to look inside and understand what each element is doing and why. Until today, they remain a popular option, especially for tabular data.

One could say that videogames have played a big role in pushing deep neural networks to their current privileged position. The need for more powerful graphics rendering hardware, created by the demand of video game fans to get higher resolution and more appealing graphics in videogames, stimulated the hardware industry to create more and more powerful GPUs (Graphics Processing Units). These devices are highly parallelized computing chips that allow for very fast computation of matrix operations. This is needed because graphical rendering engines work by applying different transformations, like translations and rotations, on elements in a (usually) three dimensional space represented by matrices.

It was only a matter of time until these devices found other uses. Specifically, they turned out to be a perfect computing device to perform operations for neural networks. One layer of a network is nothing else than a big matrix of numbers that needs to be multiplied by its input data. The overall higher computing power of modern CPUs, coupled with the recent popularity of graphics cards, quickly turned neural networks into the most popular machine learning method around, soon beating all other models in most categories. AlexNet [13] won the ImageNet competition [4] with a big margin over shallow machine learning methods in 2012, catapulting deep neural networks into the machine learning scene and positioning it as one of the most powerful ever since.

The search engine company Google has been one of the companies most actively pushing the field forward, with entire departments focused on performing research, while fully integrating the technology into their infrastructure where-ever they can. Their biggest contribution to the field, apart from research, is their open source deep learning framework TensorFlow (see Section 5.1).

One of the strongest arguments against neural networks is that they are not yet so well theoretically backed and are mostly popular due to their practical success rather than for their mathematical proofs. Many call them a black box, and are concerned about not knowing what is happening inside [12]. Efforts are being made on understanding what the inner workings of deep models are, but ultimately this problem will always remain one of the main points of critique.

The other biggest argument against deep neural networks is how fragile they are to hand-crafted attacks. Some works such as [26] describe how an image classification model can be fooled by just modifying one pixel of an image, a detail impossible to spot by a human.

2 Prerequisites and Basic Concepts

Before diving into the more complex deep learning concepts, explained in Section 3, it is important to cover the basics and fundamental concepts of the discipline.

2.1 Our New Best Friend, the Tensor

Tensors are the main mathematical structure used when working on deep learning research, and especially during implementation. Matrices and vectors are also prevalent, but it is important to note that they can easily be regarded as tensors as well, as will be discussed in the following paragraphs. Tensors will be represented with a bold typeface in this work.

A tensor can be considered to be a multidimensional array. The number of dimensions of a tensor is called *rank*, and the number of elements in each dimension (the size of the dimension), represented as an n-tuple, is called *shape*. The syntax used here for representing the shape is `(dim_1, dim_2, ..., dim_n)`, where `dim_i` is the size of the dimension i of the tensor. For example, a common $\boldsymbol{x}$ tensor for image data (stored as pixel values) has rank 4, and a shape of `(num_images, height, width, channels)`. The first dimension can be used to access each of the images, and the second and third ones to access the pixel rows and columns of each image, respectively. The term `channel` refers to the different color channels available in its representation, usually red-green-blue, so we can use it to access the three different values that each pixel contains. The rank of a tensor can also be understood as the length of the shape tuple. The total amount of elements stored in a tensor is the multiplication of all its dimension sizes.

Although vectors are commonly represented as rank 1 tensors, their natural counterpart, it is also possible to refer to column vectors as rank 2 tensors with a "dead" or "shallow" (size 1) dimension. The shape of a column vector with n entries will thus be `(n,1)`. This helps when performing element-wise operations between tensors (avoids needing to use the transpose, naturally triggers *broadcasting*, etc.), and corresponds with the mental image of a column vector, spanning along the rows, vertically.

2.2 Broadcasting

Broadcasting is the concept of adapting tensor shapes in order to perform operations between them. As an example, consider the case of having two tensors $\boldsymbol{a}$ and $\boldsymbol{b}$ of shapes `(3,2)` and `(1,2)` respectively. We would like to perform an element-wise addition on them, but clearly this is not going to work well, since their shapes do not match (i.e., it is not clear which element of $\boldsymbol{a}$ goes with which element of $\boldsymbol{b}$). The concept of broadcasting will perform an extension to $\boldsymbol{b}$, expanding its first dimension size to 3 by copying the first row two more times. Then, the element-wise addition is performed, and finally we obtain a result of shape equal to $\boldsymbol{a}$. What happened conceptually is that we added $\boldsymbol{b}$, which is a single row, to each of the rows of $\boldsymbol{a}$.

The most basic case for broadcasting is the situation in which all but one dimension matches and the sizes of the remaining dimension have a common divisor. The smaller dimension will be scaled by a certain factor in order to match the bigger one.

Broadcasting is immensely helpful when operating with tensors, as it avoids having to manually convert tensors to matching dimensions. Throughout this text, we assume broadcasting is applied whenever applicable.

This technique is a default behavior in most tensor manipulation libraries available, such as Numpy for Python.

2.3 The Family of Vector and Matrix Products

We define the **dot product** z (a scalar) of two vectors x and y as the sum of the pairwise products of their elements:

$$x \cdot y = \sum_i x_i * y_i$$

The length of x and y must match for this operation to be applicable.

We define the **matrix product** C (a matrix) of two matrices A and B as the rows by columns dot product of A and B, such that $C_{i,j}$ is the dot product of row i of A and column j of B:

$$C_{i,j} = \sum_k A_{i,k} * B_{k,j}$$

The number of columns of A must match the number of rows of B for this operation to be applicable.

We define the **element-wise product**, or **Hadamard product**, as the element-wise multiplication of the elements of A and B. Unless we consider broadcasting, the number of rows and columns of both matrices must match in order to perform this operation.

Finally, when considering tensors, one can perform any of the above products the same way they would on matrices and vectors, except for the dot product with ranks higher than 2. This situation can lead to a number of different options, and is commonly called *tensor contraction*. It will not be further discussed, since it is not needed for this thesis. Most deep learning methods involve either matrix products of rank 2 tensors, or element-wise operations on tensors of arbitrary rank.

Commonly, the term *dot product* is used interchangeably with matrix product, including most deep learning libraries. As such, we will follow this convention and use the term *dot product* from now on in this work.

2.4 Neural Networks

The goal of a Neural Network (NN) is to approximate a desired target function $f^*(\boldsymbol{x}) = \boldsymbol{y}$, that maps a data sample ($\boldsymbol{x}$) to a target value ($\boldsymbol{y}$), which is commonly a discrete label or a real valued number. During implementation, the function is commonly extended to be able to take in any number of data samples at once. In many cases, $\boldsymbol{x}$ has a shape of `(num_samples, features)`. In other words, each row corresponds to a sample, and each column to a feature of that sample. We will see later that higher rank tensors can also be used when working with sequential or image data. However, in these cases, other more advanced types of layers have to be used. The case of wanting to map a single data sample to its target is considered by supplying a shape `(1, features)` tensor. The output or target $\boldsymbol{y}$ will have shape `(num_samples, 1)` in regression tasks, and `(num_samples, classes)` in case of multi-class classification tasks, due to the use of sparse/one-hot encoded representations and probability distributions (explained in detail in Sections 2.6.2 and 2.6.3).

Each layer of an NN could be considered to be a function $f^{(i)}$, performing a mapping task between $\boldsymbol{h}_{i-1}$ and $\boldsymbol{h}_i$. Combining several of these functions together yields what is commonly known as a basic Deep Feedforward Neural Network. For example, a basic two layer NN could be $f^{(1)}(\boldsymbol{x}) = \boldsymbol{h}_1$ and $f^{(2)}(\boldsymbol{h}_1) = \boldsymbol{y}$. For n layers, this would extend to:

$$f(\boldsymbol{x}) = f^{(n)}(f^{(n-1)}(f^{(n-2)}(\dots f^{(1)}(\boldsymbol{x})\dots)))$$

When referring to the most common type of neural layer, called *dense layer*, each of the functions $f^{(i)}$ will usually compute $nonlin(\boldsymbol{h}_i \cdot \boldsymbol{w}_i + \boldsymbol{b}_i)$. Here, *nonlin* is a non-linear function or activation function (see Section 2.5), $\boldsymbol{w}$ is the weight tensor of the layer, and $\boldsymbol{b}$ is the bias tensor.

Assuming we are working on a classification task, and considering the case of a single layer network, observe how the shape of $\boldsymbol{w}$ is forced to be `(features, classes)` in order for the dot product to return the shape we desire:

$$\texttt{(num\_samples,features)}\cdot\texttt{(features,classes)}=\texttt{(num\_samples,classes)}.$$

The true power of NNs arises when we chain several layers after each other. Intermediate layers will have their own number of neurons, which will be reflected on the shapes of the intermediate tensors that are generated. Consider the case of a two layer network. We can call the intermediate tensor $\boldsymbol{h}$, the result of processing $\boldsymbol{x}$ through the first layer, and being input for the second layer. Let `hidden` be the number of neurons that we chose for the intermediate layer. Then $\boldsymbol{h}$ will have a shape of `(num_samples, hidden)`. The weights $\boldsymbol{w}_1$ of the first layer, used to obtain $\boldsymbol{h}$, are thus forced to have a shape of `(features, hidden)`. Likewise, $\boldsymbol{w}_2$, used to obtain $\boldsymbol{y}$ from $\boldsymbol{h}$, will have a shape of `(hidden, classes)`.

2.5 Activation Functions

An activation function is a nonlinear function applied to the output of a layer, prior to passing it over to the next layer as input. Historically, neural networks commonly used the so called *sigmoid* function.

Its main characteristic is that it squashes all values in the $(0, 1)$ range. The input value 0 will correspond to the middle value of the function, 0.5. See Figure 3 for a plot and formula of the function.

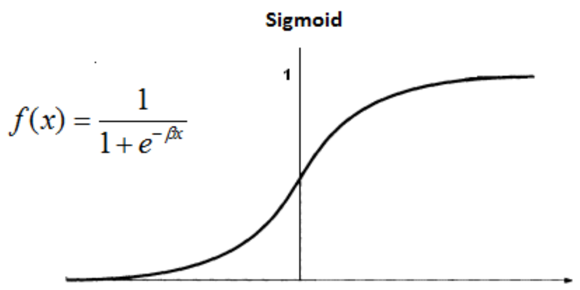


Figure 3: The *sigmoid* activation function.

As the field evolved, researchers have found that other activation functions are more useful and yield better results. The most common now is the rectified linear unit (*ReLU*) [16].

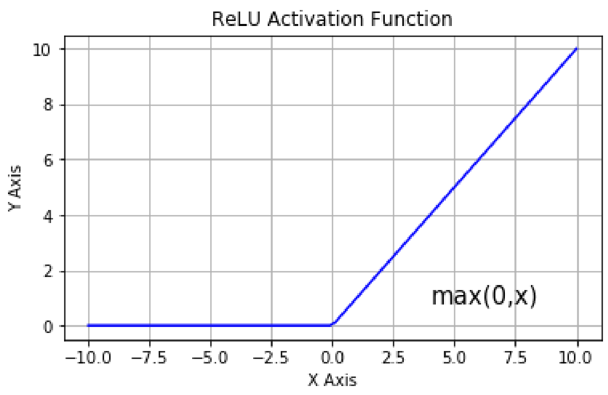


Figure 4: The rectified linear unit activation function.

All negative values are transformed into a 0, while all positive values remain untouched (see Figure 4). Some argue that it approximates the behavior of real neurons slightly better, and present interesting qualities for deep neural networks.

Other activation functions commonly used include the *hyperbolic tangent* (see Figure 5), which is similar to the *sigmoid* but has a range of $(-1, 1)$; and the *leaky ReLU*, a modified version of *ReLU*:

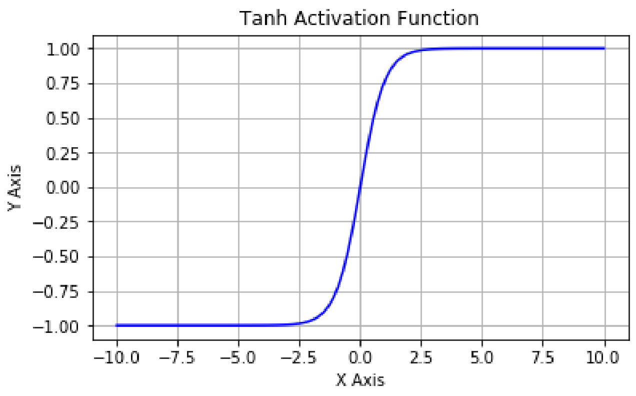


Figure 5: The hyperbolic tangent activation function.

$$f(x) = max(0.1 * x, x)$$

This modification allows for negative values to have a non-zero gradient. This function is an attempt to solve the issue of “dead neurons” that can arise when using *ReLU*, since the negative side of the function always yields a 0, resulting in a flat gradient. Once a neuron is stuck in this area of the function, it can no longer update its weights. Gradients and weight updating are discussed more in depth in Section 2.7.

Another important activation function is the *softmax* function:

$$f(x) = \Big(\frac{e^{x_1}}{\sum_j e^{x_j}}, \frac{e^{x_2}}{\sum_j e^{x_j}}, \dots, \frac{e^{x_n}}{\sum_j e^{x_j}}\Big)$$

It behaves like a smooth approximation of the *argmax* function with one-hot encoded output (which returns a vector with the same length as the input, and holds 0’s everywhere except for the position of the maximum element, which contains a 1. Softmax produces a probability distribution as output, with the sum of all output values equal to 1, and the highest value in the input vector will convert into a value close to 1 in the output. It is commonly only used on the final layer of the network, in conjunction with a *cross-entropy* loss (see Section 2.6.3). It differs from most other functions in that it acts on all outputs together, instead of individually (i.e., each value depends on all the other values).

The reason why we need activation functions lies in the fact that if we only apply linear combination operations (matrix multiplications), higher levels of abstraction cannot be obtained. Consider the situation where we have three layers with their respective weights $\boldsymbol{w}_1$, $\boldsymbol{w}_2$ and $\boldsymbol{w}_3$. Ignoring the bias terms $\boldsymbol{b}$, let us consider what happens if we progressively apply layers without activation functions in between:

$$f(\boldsymbol{x}) = \boldsymbol{x} \cdot \boldsymbol{w}_1 \cdot \boldsymbol{w}_2 \cdot \boldsymbol{w}_3$$

This result could have been equally obtained through a single layer, by using an associated set of weights $\boldsymbol{w}_T$ such that:

$$\boldsymbol{w}_T = \boldsymbol{w}_1 \cdot \boldsymbol{w}_2 \cdot \boldsymbol{w}_3$$

The nice properties of deep neural networks with several layers vanish, as all our layers collapse into one if we do not use activation functions. By using a non-linear function between the matrix multiplications, we ensure that there is no equivalent set of weights $\boldsymbol{w}_T$ that could possibly imitate the behavior of several layers.

2.6 Loss Functions

The loss function is a metric applied to the predictions of a neural network, in order to judge how much they differ from the true labels. Such a function is needed since a neural network is nothing else than a complex optimization problem. By defining a loss function, we effectively have a target that we can minimize using optimization tools. The algorithms employed to train a neural network are all based upon the principles of gradient descent (more on this in Section 2.7).

2.6.1 Regression

For regression, one of the most commonly used loss functions is the squared difference, also called mean squared error. Given a vector of predicted values y_p and a vector of true labels y, one wants to know how much they differ, and punish quadratically more those that are especially far away.

$$L(y, y_p) = (y - y_p)^2$$

This is also the function that is commonly minimized in the basic regression analysis technique *least squares.*

2.6.2 One-hot Encoding

In order to proceed to talk about the most common type of loss function used for classification problems, it important to properly define the concept of *one-hot* encoding.

When working with classification labels, one is usually given a vector where each value indicates the class of the corresponding data point in $\boldsymbol{x}$. Given m different classes, the values in this vector usually range between 0 and $m-1$.

In order to one-hot encode this labels tensor, a new sparse tensor of shape `(num_samples, classes)` will be created. Each row of this new tensor will have all zeros except for the column corresponding to the class of that row, where a 1 will be stored (hence the term "sparse"). In this manner, the 1's are indicating through their column index the class that each data sample corresponds to. The usefulness of this representation will be made clear in the next section.

2.6.3 Classification

When classifying with neural networks, it is important to maintain balanced class distances. This means that the numerical distance between class representations shall be equal.

If class labels are stored as a number (from 0 to $m-1$), and the network is trained to predict a single output number, then the above restriction does not hold. The distance between the first and the second class is not the same as the distance between the first and the last one, for example. The loss for a missclassification will vary depending on the specific classes that are being considered. This goes against the intuition that a missclassification should be handled equally, no matter which two classes are being compared.

To solve this issue, a network is usually trained to output m different values, passed through a softmax activation function, effectively producing a probability distribution for the classes. The shape of such output will then commonly be `(num_samples, classes)`, matching the shape of the one-hot encoding of the labels. Those encoded labels can now be interpreted as the true probability distribution for each sample. The probability of the correct class should be 1 and all other classes should have a probability of 0. What has to be done now is to somehow judge the badness of the prediction, or in other words, the amount of difference between the prediction and the true distribution.

To do this, a typical approach is the cross-entropy metric, defined formally as follows:

$$-\sum p_i * \log(q_i)$$

where p and q are two vectors of the same length representing discrete probability distributions. This metric effectively measures how different two probability distributions are, being a perfect fit for the problem at hand.

2.7 Optimization Process

Once the model is defined, all weights are initialized (usually with random normal distributions), and the correct loss function is set, all that is left to do is update the weights to achieve the desired behavior and performance of the network. For this, the gradient of the loss with respect to all the different sets of weights of the network has to be obtained, and a step towards the negative gradient has to be taken for all of them.

The way most deep learning frameworks perform this task is by keeping an internal graph representation of the model that was built. For each operation that is supposed to be performed on the input tensors, the derivative of that operation is also present. During training, the algorithm of back-propagation uses these derivative functions to compute the local gradient of each node in the graph, and accumulates the upstream gradient that is flowing up through the graph until it reaches all weights.

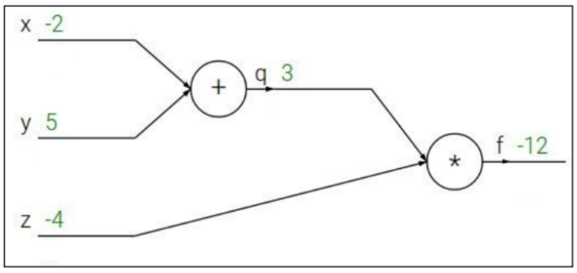


Figure 6: An example of a computational graph.
Source: `http://cs231n.github.io/`

As a trivial example, consider the graph of Figure 6. The inputs x, y, z are processed through the nodes of the graph, representing operations, until the final result f is produced. The algorithm of backpropagation will start with the final node, and will keep track of the upstream gradient. This is the accumulated gradient that has been computed so far. Recall that the chain rule for derivation of a composed function $z = g_1(y); y = g_2(x)$ works as follows:

$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y}\frac{\partial y}{\partial x}$$

In the above diagram, to obtain the gradient of f w.r.t. y, one could compute:

$$\frac{\partial f}{\partial y} = \frac{\partial f}{\partial q} \frac{\partial q}{\partial y}$$

The algorithm will consider $\frac{\partial f}{\partial q}$ to be the upstream gradient for the plus node that produces q. The local gradient of the node, $\frac{\partial q}{\partial y}$, is multiplied with the upstream one to produce the final gradient for input y. To complete the example, let us calculate the final gradient for all inputs of this graph. Recall that $f = q * z$ and $q = x + y$.

$$\begin{aligned} \frac{\partial f}{\partial z} &= q = 3 \\ \frac{\partial f}{\partial y} = \frac{\partial f}{\partial q} \frac{\partial q}{\partial y} &= z * 1 = -4 \\ \frac{\partial f}{\partial x} = \frac{\partial f}{\partial q} \frac{\partial q}{\partial x} &= z * 1 = -4 \end{aligned}$$

Once the gradient for each set of weights is obtained, they are updated following a simple rule:

$$w = w - lr * \frac{\partial L}{\partial w}$$

Here, w is a single weight entry, and L is the loss. The learning rate lr decides how big of a step is taken each time. The choice of a good learning rate makes or breaks the training procedure, and no flawless method is known for choosing a right learning rate, other than by trial and error.

2.7.1 Stochastic Gradient Descent

Until now, it has been assumed that the whole dataset is passed through the network to obtain the gradient. However, doing this is very computationally inefficient. Modern gradient descent algorithms always make use of smaller batches of data. When computing the gradient by using only a subset of the data, it is certain that the step taken will not follow the global steepest descent with precision. However, a certain level of randomness has actually been found to be beneficial for the learning procedure. The overall direction of learning will still be accurate, and by taking stochastic steps instead of perfectly accurate ones, local minima can be avoided.

This is also the origin of the name *batch* that is commonly used to refer to the first dimension of the training and output data.

2.7.2 Adam

Adam (from adaptive moment estimation) is currently one of the most popular and generally recommended optimization algorithms [21]. It draws from the benefits from two other well known algorithms: *AdaGrad* (adaptive gradient algorithm) and its improvement *RMSProp* (root mean square propagation).

They both maintain internal per-parameter learning rate, allowing a more fine-grained control over the speed at which each weight is updated. The second one has its parameters adapted at each iteration by using the average of recent magnitudes (called the *raw* or *first moment*) of gradients for each weight. Noisy data commonly exposes high variations in gradients, something that this method is able to overcome by smoothing out those movements.

The main difference resides in how Adam adapts the learning rates. Instead of only using the first moment, as RMSProp does, it also uses the *central* or *second moment*, which takes into account the variance of the data. Without going into much detail, this helps to keep a more intelligently updated set of learning rates for gradient descent, overall improving performance.

This method was chosen to train the neural network of this project.

3 Advanced Deep Learning

After having covered all the basic building blocks of Deep Learning, this section now describes the advanced neural architectures that are most used today.

3.1 Recurrent Networks

When processing sequential data, one finds basic neural networks lacking a key component: sequential understanding. Imagine working with sentences, each word being encoded as a number. Regardless of the task at hand, those words need to be fed into the network to process their meaning. However, a basic deep forward network can only receive the whole sentence at once, losing its sequential meaning. A better way to do this is to use *recurrent neural networks* (RNNs). Their input, instead of being the usual `(batch, features)`, will now be `(batch, sequence, features)`.

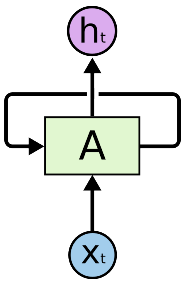


Figure 7: A basic recurrent neural network.
Source:
`http://colah.github.io/posts/2015-08-Understanding-LSTMs/`

The main idea behind this type of network is having an internal state that is updated as we progress through the sequence. As it can be appreciated in the graphical representation of an RNN (see Figure 7), there is a main loop connecting its inner state with itself. Also, we find an input value $\boldsymbol{x}_t$ and an output value $\boldsymbol{h}_t$.

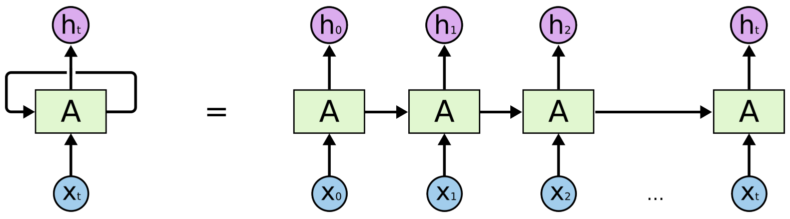


Figure 8: Unrolling an RNN.

If we unroll the network to accommodate a sequence of t inputs (such as a sentence with t words) then we would find ourselves having something similar to the NN depicted in Figure 8.

By observing a very basic example of a network that is working on predicting the next character in a sequence (see Figure 9), one can understand the inner workings of an RNN. What we here call *hidden layer* is the internal state of the network. To go from input to hidden layer shape, we use the weights $\boldsymbol{w}_{xh}$. To transition from the last inner state to the next, we use the weights $\boldsymbol{w}_{hh}$. Finally, to use this inner state to produce an output we use the weights $\boldsymbol{w}_{hy}$.

It is assumed that the vocabulary for this task only contains 4 letters: $\{h, e, l, o\}$, represented using one-hot encoding, as it can be appreciated in the red boxes. After applying the function associated to the weights $\boldsymbol{w}_{xh}$, the first hidden state is produced in the first green box. This inner state hopefully captures the fact that an h has been read. By using this state, an output probability distribution can be generated by applying the function associated to weights $\boldsymbol{w}_{hy}$. In this example, the blue boxes have not yet suffered the application of the *softmax* function, as this step is assumed

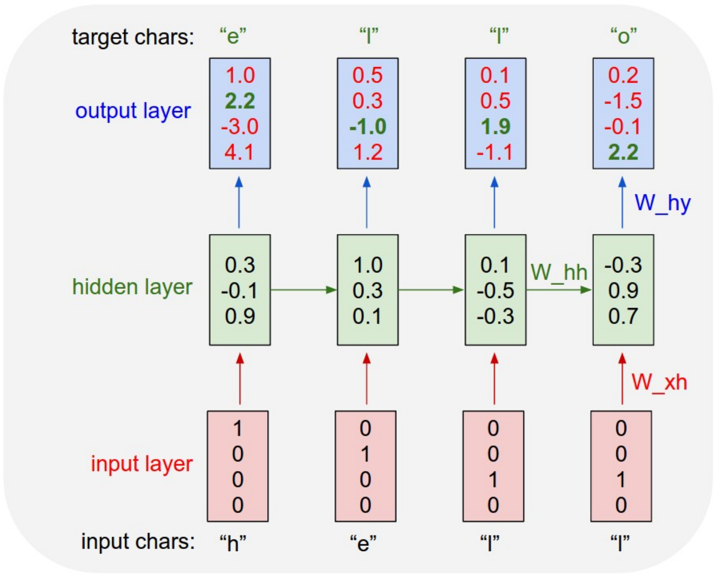


Figure 9: Predicting characters in a sequence.
Source:
`https://karpathy.github.io/2015/05/21/rnn-effectiveness/`

to be performed inside of the loss function. The correct target letter, *e*, is highlighted in green. However, the network has predicted *o* as the target at this step, since its index has the highest value (4.1). The next hidden state can now be computed by adding the last hidden layer to the result of the input function, and the cycle repeats itself until the last output is finally reached. Progressively, the hidden state is required to capture the information of not only the last input, but hopefully several of the previous inputs as well, as they are probably useful for predicting the next target.

In order to train a recurrent neural network, it has to be "unrolled" in time, since the information at all time-steps has to be used to compute the gradient. Some issues such as vanishing or exploding gradients [17] can arise. This means that the gradient values decrease or increase exponentially as they backpropagate through the network. When they finally reach the upper layers, their values are either too large or too small, producing wrong updates to the weights. Usually, this can be alleviated by using gradient clipping, which is basically an upper threshold that the gradients are not allowed to grow beyond. Whenever a value surpasses the threshold, it is "clipped" back to the maximum permitted value.

The two RNN types most used today are LSTM (Long Short Term Memory) and GRU (Gated Recurrent Unit). They both contain an internal gate mechanism to decide, at each timestep, which values in their inner states should be kept and which forgotten. Although LSTMs are more complex and in theory should be able to capture temporal information better, in practice GRUs perform very similarly and consume much less computational power.

CTC Loss: Connectionist Temporal Classification

The Connectionist Temporal Classification loss is a function that evaluates performance of a model that is predicting a sequential output. Its two most common use cases are speech recognition and OCR.

The inherent difficulty of evaluating these types of tasks is caused by the concept of alignment. Suppose we have some input data such as audio waves, and an output such as the text corresponding to that audio wave. There is no way to know which segment of the waves corresponds to each character of the text. This lack of information makes training and evaluating these types of models difficult.

Suppose the model has learned to recognize the sound “o”. Two instances of waves for the word “hello” could look quite different depending on how long the segment for the “o” sound is. It would be natural to believe that the model will attempt to produce more “o” characters the longer the sound is, and should not be punished for this. The CTC loss attempts to close the gap that arises from the lack of alignment, and does not punish these length differences.

The basic idea behind this algorithm is to eliminate character repetitions, and remove the special empty characters (ε) that the model is encouraged to generate. Once this removal is performed, the output is compared with the true label to judge performance. Consider the following example output: $heel\varepsilon looo$. After performing character repetition removal, we obtain $hel\varepsilon lo$. Finlly, by removing the empty character, we end up with the target word, *hello*. Any variation of the output will be accepted as long as the algorithm is able to generate the target. However, outputs such as $he\varepsilon ello\varepsilon o$ are not correct (the final output would be *heelloo*), and will thus yield a higher loss. Note how the purpose of the empty character is to allow for the production of repeated characters, such as *ll* in hello, which would not be possible to produce otherwise.

Finally, it is worth mentioning that all functions used in the algorithm have to be differentiable in order to compute the gradients necessary for backpropagation. The algorithm will not be covered with more detail in this work due to its complexity, but there is an excellent article on DistillPub [7] shedding more light on the topic.

3.2 Convolutional Networks

Locality abstraction is another property where dense neural networks fall short. Suppose we would like to classify images based on the objects that are depicted in them. We would like the network to find these objects no matter where they are located in the image. This is a challenging task that dense neural networks are commonly not able to resolve comfortably, often relearning to detect these objects separately for each location on the image. A different type of network, called convolutional neural network (CNN), is usually used for this type of task. They are able to abstract away locality and learn patterns for object detection regardless of their position within the image.

The basic procedure carried out by a CNN is the application of a series of kernels on the image, by sliding a window of certain size across it. A kernel, also called filter, is a small rank 3 tensor whose job is to find patterns in the data. Recall that image tensors also have rank 3: `(height,width,channel)`. The kernel usually spans across all channels (depth of the image), but only covers a few pixels in height and width at a time, effectively representing a small crop of the image. At each position, the kernel is multiplied element-wise with the window and summed up to a single value. This value represents the presence of the specific pattern at that specific location. As an example, please refer to Figure 10.

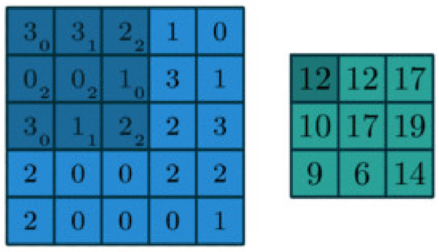


Figure 10: The basic convolution operation.

Here, an image tensor of shape `(5,5,1)` is considered in blue. Note the absence of depth (the `channel` dimension is 1), as would be the case with gray-scale images. The kernel, of shape `(3,3,1)`, is slid over the image, moving in steps of 1 pixel across the whole image, both horizontally and vertically. The values of the kernel are represented in the lower right corner of the grayed out cells, which correspond to the current window position. After an element-wise multiplication and an addition across all axes, the value 12 is produced, which is stored in the first slot of the green rank 2 output tensor. Note that this tensor will have rank 2 regardless of the presence of several channels, as the final sum in each step flattens the result into a single value. For this image and kernel, a total of 9 windows are captured, and the resulting shape is `(3,3)`.

Usually, a convolutional layer possesses several kernels (such as 16 or 32), each of which producing a new output tensor like the one above. All these outputs will be stacked along a new third dimension, which could be regarded as a new channel dimension. If the example of Figure 10 had 16 kernels, the final output shape of the layer would be `(3,3,16)`.

We call *kernel size* the height and width tuple defining the size of the window, *stride* the number of pixels that the window is moved each time, and *padding* the number of pixels added to the border of the image to compensate for the window size. This last parameter is commonly used to preserve the original image height and width of the input data as more and more layers are applied. In our example, a padding of 1 would allow to slide the windows 5 times in each axis, producing an output tensor with the same height and width as the input. The added values are commonly zeros.

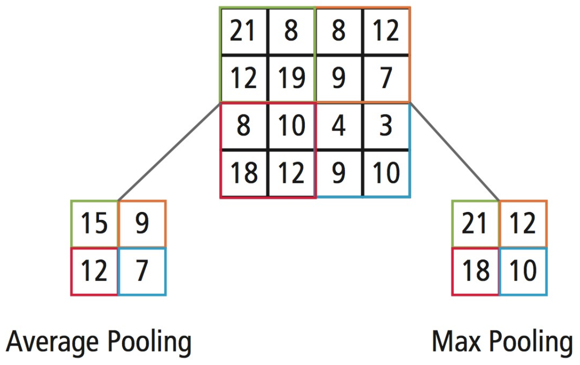


Figure 11: Two types of pooling operations.
Source: `https://www.embedded-vision.com/platinum-members/cadence/embedded-vision-training/documents/pages/neuralnetworksimagerecognition`

Although progressively reducing the size of the intermediate tensors is desirable, there is a technique that is generally better suited for this task: the *pooling layer*. Consider the outputs of a convolutional layer to be indicators of presence of patterns in the image. A pooling layer attempts to reduce the size of this output while keeping the essential information. The most common operation performed in such a layer is a *max* function over a sliding window of shape `(2,2)` and stride 2. Consider the example depicted in Figure 11.

Each block of `(2,2)` pixels will be converted into a single pixel by applying a *max* function over them. Commonly, *max* is preferred over other functions such as *average* since we would like to keep the information regarding the maximum presence of certain patterns. Surely, the values surrounding a high value are activations of the same pattern that have been captured in a slightly miss-aligned manner, thus only returning a partial activation value. By using the *max* value, we make sure to keep the whole pattern presence information, and avoid the progressive dilution that would occur if we took averages.

3.3 Regularization

Regularization refers to the concept of controlling or restricting the growth of the parameters of a machine learning model to avoid overfitting. This phenomenon occurs when a model attempts to reduce the error over the given dataset to a point where it is not generalizing well anymore, returning a high error on values outside of the training samples. Depending on the type of model, different techniques exist to tackle this problem.

Simple models such as linear regression and logistic regression restrict the growth of the weight vector w by adding a new term to the loss function, that could look like the following:

$$\sum_i w_i^2$$

This term penalizes large weight values, which are usually necessary for the model to develop overfitting behavior. Since the optimization process involves reducing the loss, a balance between training set accuracy and weight complexity will naturally emerge.

Deep learning models can also utilize this type of regularization. However, its usage is not as common as some other available techniques.

The most important and arguably most basic type of regularization that one can use is *early stopping.* During the optimization process, one keeps track of both the training set loss as well as a hold-out validation loss, ideally also tracking other metrics such as accuracy. Once the validation loss has stopped improving, the training is manually stopped. This is the inflexion point where training has been capturing the generalizable properties of the data, and from which on overfitting would occur, worsening performance on new data.

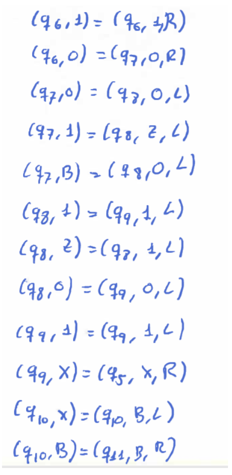
The second most important regularization method in this field is *dropout* [25]. Usually regarded as a layer, dropout will randomly zero out a certain percentage of the input values. The next layer will thus receive incomplete data. As training progresses, this random dropout of values will help the network find patterns in the data that do not require the specific presence of a certain small number of values. Rather, it will attempt to find more overarching properties, which are not easily influenced by the absence of a certain percentage of values. It is especially common to use this type of regularization in CNNs, but it can be used with any other type of network as well.

4 Case Study

This section describes the problem that this project aims to solve, by introducing the data that was used and the model that was built to process it.

4.1 Turing Definition Lines

The source material for the dataset used are scanned exams of previous students of a course at the University of Cantabria, in PDF format. These PDFs contain not only Turing machines, but also other exercises, numbers, headers, notes, etc. The first step is thus to crop down these PDFs into images that contain, to the maximum degree possible, only Turing machine definitions. One such crop may look like the Figure 12.



$(q_6,1)=(q_6,1,R)$
$(q_6,0)=(q_7,0,R)$
$(q_7,0)=(q_7,0,L)$
$(q_7,1)=(q_8,2,L)$
$(q_7,B)=(q_8,0,L)$
$(q_8,1)=(q_9,1,L)$
$(q_8,2)=(q_7,1,L)$
$(q_8,0)=(q_9,0,L)$
$(q_9,1)=(q_9,1,L)$
$(q_9,X)=(q_5,X,R)$
$(q_{10},X)=(q_{10},B,L)$
$(q_{10},B)=(q_{11},B,R)$

Figure 12: A crop of a Turing machine definition.

The corresponding labels or target data of this dataset are, naturally, the transcriptions of these Turing machines. Although the professor in charge of evaluating these exams has manually transcribed them all for evaluation, the lack of alignment as well as the change in style and syntax hindered the usage of this resource. Since a specific syntax is not strongly enforced in the course, students often adopt one of several options available, or come up with their own. The simulation script that evaluates the Turing machines requires a specific syntax, so the professor has had to perform syntax conversion dynamically while transcribing the exams. As a result, a new dataset has been created by manually relabelling crops of these exams, as will be detailed further below.

Once the crops have been obtained, the open source tool Tesseract[2] is used to predict bounding boxes around pieces of text. These predictions can be more or less accurate depending on the specific crop, its layout, the separation between characters, the style and clarity of the student's handwriting, etc. Around 70% of the boxes were accurate, and have been used to build the dataset. One such predicted box image can look like Figure 13.

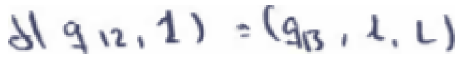


Figure 13: A predicted box for a line of a Turing definition.

On top of purging out those boxes that were not accurate, an attempt has been made to keep a certain balance with respect to the number of boxes kept per exam. A total of 61 crops have been processed, associated to around 30 different exams, and the number of automatically recognizable Turing definition lines in each of them differs greatly. Thus, for crops with an extremely high number of boxes, a certain proportional amount was removed to keep a balanced number of examples per handwriting style. Finally, while performing this removal, it became apparent that the proportion of symbol appearance should also be taken into account. If always removing the last boxes, then numbers from 4 6 onward would become much less frequent in the dataset, since they tend to appear later on in the definition of the Turing machine. Keeping a healthy class balance in the dataset is important, so removing random lines seemed to be a good approach.

Once enough of these boxes have been generated (each as its own independent image), they had to be manually labelled. For the initial version of the dataset, 400 images were kept. Each file was given a name in the form of `cropnum_boxnum`, where `cropnum` is the *id* of the source crop, and `boxnum` is used to differentiate between boxes of the same crop. A separate file holds the labels corresponding to each filename. The format looks like the following:

```
1_2:  d(q12,1)=(q13,1,L)
1_8:  d(q1,1)=(q1,1,R)
1_11: d(q1,1)=(q1,1,R)
```

[2] https://github.com/tesseract-ocr/tesseract

```
1_13: d(q13,1)=(q13,B,L)
1_15: d(q13,1)=(q13,B,L)
1_16: d(q1,0)=(q2,0,R)
```

It is then possible to load all images and this text file, and parse it to obtain the labels for training.

4.2 Improving Number Recognition

After analyzing results of the initial versions of the model, a major weakness could be appreciated in number recognition. The number of samples is probably not high enough for the model to properly learn to distinguish between numbers correctly. Many of the other characters, including parenthesis and commas, can be predicted out of context (position in the sequence). However, although the model can easily learn to predict when a number should appear, distinguishing which number it is is a more difficult task. By generating synthetic sequences with a variety of numbers in them, the model has a chance to learn to better distinguish numbers.

To generate this dataset, samples from the MNIST dataset [14], a popular and widely available handwritten digit recognition dataset, were randomly placed together in images containing sequences of 4, 5 and 6 numbers, automatically labelling at generation time. One such sequence is displayed in Figure 14.

Figure 14: A synthetic number sequence with label *336920*.

This synthetic dataset has been used together with the Turing lines dataset to train the model, adding about 600 new images to the set.

4.3 Preprocessing

Before training, it is necessary to transform the images into arrays of numbers, shaped adequately for the neural network to be able to read and work with them. In the following paragraphs, this preprocessing step is further detailed.

First, all images were resized to a width of 170 and a height of 30. These numbers strike a balance between size (a higher number considerably affects the computational resources necessary to train the model) and clarity of the images (which influences accuracy). The original average size was 330x60.

Then, all images are transposed to invert the width and height axis, a necessary step that will be explained in detail in Section 4.4.

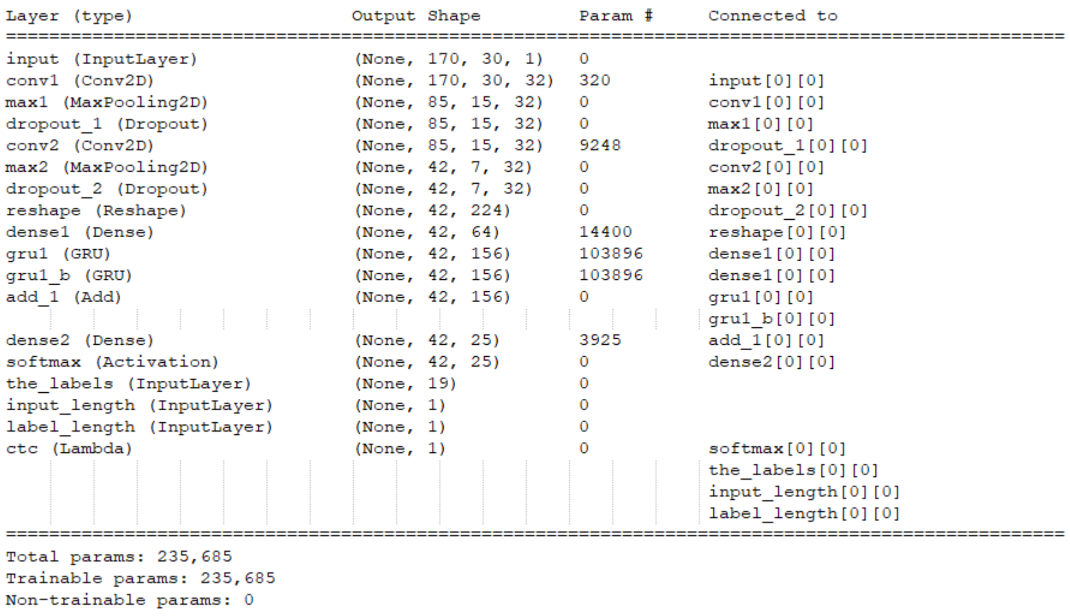
Since color is not a valuable parameter for this project (it is irrelevant which pen color the student chose to write the exam), images are loaded in black and white. Thus, each image is composed of `width x height` values. Usually, images of this type are encoded using values between 0 and 255, representing gray-scale values from black, the lowest, to white, the highest. However, it is common practice in machine learning to scale down values to a range of $(0, 1)$, or sometimes $(-1, 1)$. The former was chosen for this project, so all pixel values were divided by 255. This strategy is called normalization and is most useful when the ranges of different attributes differ. Attributes with a higher range or higher average values produce a higher impact on neuronal activations, regardless of the actual usefulness of that attribute. In our case, all attributes have the same range, but the convention is followed for clarity.

Finally, although the model is capable of learning from the data as is, the pixel values of the image have been inverted, effectively turning the background black and the handwriting white. This way, only those pixels actually containing useful information (the strokes of the symbols) have high values, while the background has values close to 0. To perform this operation, it is merely necessary to replace each pixel value x with a new value $1 - x$, as long as this is done after the re-scaling procedure of the preceding paragraph.

4.4 The Model

The model that has been used in this project to perform OCR over the Turing definition lines is based on the CRNN model [24]. The key ideas and insights of this model will be explained along with its structure in the following paragraphs.

The basic idea behind this architecture is to combine the locality abstraction capabilities of CNNs with the sequential understanding capabilities of RNNs. The head of the network is a CNN working on extracting image attributes. In this case, it will capture the presence of specific symbol constituents, which can later be used to judge whether a specific symbol is present or not. Then, a dense layer connects these attributes to one or several RNNs, which will read the image attributes column by column. The hope here is that the RNN learns to predict the target sequence by looking for the presence of symbols from left to right, as well as by having contextual information regarding the symbols that have already been seen.



```
Layer (type)                    Output Shape         Param #     Connected to
==================================================================================================
input (InputLayer)              (None, 170, 30, 1)   0
conv1 (Conv2D)                  (None, 170, 30, 32)  320         input[0][0]
max1 (MaxPooling2D)             (None, 85, 15, 32)   0           conv1[0][0]
dropout_1 (Dropout)             (None, 85, 15, 32)   0           max1[0][0]
conv2 (Conv2D)                  (None, 85, 15, 32)   9248        dropout_1[0][0]
max2 (MaxPooling2D)             (None, 42, 7, 32)    0           conv2[0][0]
dropout_2 (Dropout)             (None, 42, 7, 32)    0           max2[0][0]
reshape (Reshape)               (None, 42, 224)      0           dropout_2[0][0]
dense1 (Dense)                  (None, 42, 64)       14400       reshape[0][0]
gru1 (GRU)                      (None, 42, 156)      103896      dense1[0][0]
gru1_b (GRU)                    (None, 42, 156)      103896      dense1[0][0]
add_1 (Add)                     (None, 42, 156)      0           gru1[0][0]
                                                                 gru1_b[0][0]
dense2 (Dense)                  (None, 42, 25)       3925        add_1[0][0]
softmax (Activation)            (None, 42, 25)       0           dense2[0][0]
the_labels (InputLayer)         (None, 19)           0
input_length (InputLayer)       (None, 1)            0
label_length (InputLayer)       (None, 1)            0
ctc (Lambda)                    (None, 1)            0           softmax[0][0]
                                                                 the_labels[0][0]
                                                                 input_length[0][0]
                                                                 label_length[0][0]
==================================================================================================
Total params: 235,685
Trainable params: 235,685
Non-trainable params: 0
```

Figure 15: Summary of the model layers.

The table of Figure 15 displays the output shapes of all layers in this model, as well as the total number of trainable parameters. The batch size is represented as `None` because no specific size has been given for this dimension, reflecting that any number of samples can be processed at once, without restrictions other than memory limitations.

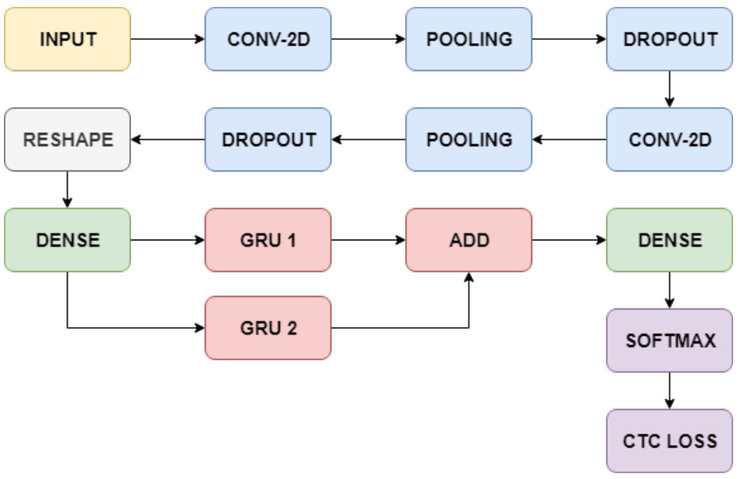


Figure 16: The graph of the whole network.

Figure 16 shows the connections between the layers. Blue represents the convolutional segment, and red the recurrent segment. Green is used for the two dense layers. First of all, it is necessary to transpose the images and feed them into the network in the format `(width, height)`, instead of the usual `(height, width)`. Recall that RNNs, ignoring the batch dimension, expect an input shape of `(sequence, features)`. In our case, we would like to consider each column of the image to be one step in the sequence. Therefore, the `width` dimension has to come first, since it is the one that can be used to access each of the different columns present in the image.

By looking at the table it can be observed that the first convolutional layer fills in the channel dimension with 32 filters, which will be kept until reaching the recurrent segment. Max pooling and dropout are then applied to the result, and the process is repeated, effectively dividing both height and width by 4. Recall that max pooling keeps the maximum value in each 2x2 tile, and dropout randomly zeros out a certain percentage of values to increase the robustness of the model. Before plugging this intermediate tensor into an RNN, it is passed through a dense layer to reduce its size. This layer takes the output of the convolutional pipeline and maps it onto a smaller vector, learning to capture and compress the important information. Consider that, without this reduction, the recurrent layer would be receiving `num_rows * num_filters` values at each timestep, which would amount to 224 with current parameters. Most probably, some of this information is redundant or unnecessary. Keep in mind that in each of these timesteps, the presence of high level patterns, ideally whole characters, has to be encoded. Each neuron is capable of representing one of these patterns, and it is unlikely that such a high number of them is required for this task.

The recurrent segment is composed of two parallel GRU layers. This means that both will, independently, read the same inputs and produce a separate set of outputs, the only difference being that the order of the inputs is reversed for one of them. The outputs of the natural and the reversed GRU are finally added together. The hope is that each of them will be able to capture different sequential patterns that would have been otherwise impossible to find. Although theoretically all necessary information is present in the normal sequence, it has been shown that in practice, bi-directional recurrent layers help to improve performance [23]. Each of the two parallel layers will receive a column at a time, and update their inner state accordingly. At the same time, they will generate an output that aims to convey character presence information for that column. Recall that, even if two consecutive columns are capturing information of the same character in the image (say, because the character is rather wide and spans several columns), then the network is going to predict duplicates. However, the CTC Loss function is going to take this into account and will not punish the model for doing so. The inner state of the recurrent layer is valuable for this task since knowledge of the previous characters can greatly help towards judging what might be the next character. For example, after having opened a '`(`' character, it is very unlikely that another appears. Likewise, if one appeared, then one of the last characters of the sequence will be the '`)`' character.

Finally, one last dense layer followed by a softmax activation is applied to reduce the output of the RNN at each timestep to one accommodating the length of the vocabulary. This is because the goal is to produce a sequence of probability distributions over the vocabulary. Currently, for this project, 25 different symbols can be found in the labels of the dataset, which are the following:

```
( ) , 0 1 2 3 4 5 6 7 8 9 = B C F L R X Y Z d q
```

The remaining `InputLayer`s that can be seen in the table are required for the `Lambda` loss function, which in this case is the *CTC Loss*, as has been described in Section 3.1.

4.5 Metrics

It is always recommended to keep track of at least one meaningful metric of your model, not only to judge the progress during training, but also to compare it with the test set performance. The metric should be able to express the quality of the predictions in a human-understandable way.

The percentage of hits (usually called accuracy) in a classification task, for example, is useful in the sense that it allows you to judge the probability of the model to make a mistake when you ask it for a new prediction. Other metrics such as precision and recall are useful in situations such as binary classification, and allow to judge not only how likely the model is to be correct, but also if the mistakes it makes are due to overconfidence or the lack of it.

For this project, such metrics are not very useful. The output of the model is a string of characters that should match as much as possible the desired target string. However, just judging whether the string matches or not is not a good metric, since a string that only contains one wrong character is much more desirable than one where most of them are wrong. As such, a metric that can take into account the amount of difference between two strings is desired. The most common metric for this situation is the edit distance. This is, the number of character edits (insertions, deletions, replacements) that are required to change one string into the other. This was the chosen metric to evaluate this model.

4.6 Parameter Tuning

Training a complex network such as this one is a challenging task with lots of trial and error required. Tuning the hyper-parameters until a satisfying result is attained can prove to be surprisingly difficult.

Oftentimes, a plateau effect was experienced during training, in which the model will refuse to advance from a certain point, not being able to reduce the loss any further. In previous training sessions the model had shown the capacity of improving past that point, so one option to consider in this situation is that it might be stuck in a local minima. Through lots of experimentation, three key parameters that one needs to change to overcome this plateau effect were pinpointed: the learning rate, the batch size, and the optimization algorithm. A higher learning rate may help the optimization algorithm to “jump” out of the minima and continue training. A lower learning rate may help the model find a “hole” which was too small to enter with the previous rate, through which it can descend and continue improving. Batch size is a factor that heavily influences the type of step we take when updating our weights. Too low batch sizes result in wrong steps, often mistaking peculiarities of the specific data points for the true direction that should be taken. Too high would reduce the stochasticity that is usually desired. Finally, different optimization methods use different update rules and carry certain momentum parameters, which can heavily influence the type of step that is taken. By carefully adapting all of these parameters, it is usually possible to break through the plateau effect.

The *Adam* optimizer function was chosen for this project (see Section 2.7), as is it commonly recommended and it helped avoid the plateau effect more than other methods.

When using a Google Colaboratory[3] GPU instance and GPU compatible layers (see Section 5.2), it is possible to train the model with the initial dataset in about 30 minutes. If only the CPU is used, this drops to about one hour. For this benchmark, early stopping was used as explained in Section 3.3, with training being considered finished when the test set metric stops improving.

The training loss can be seen decreasing in the graph of Figure 17. The jig-saw pattern is due to how data augmentation was used. Every 5 epochs, a new transformed dataset is generated dynamically, thus the model sees a slight loss increase when it receives the new data. The transformations performed are rotation, shearing and zoom, all in small quantities. This helps towards generalization, and avoids that the model learns patterns through precise location or shape, instead promoting generalization.

[3]`https://colab.research.google.com/`

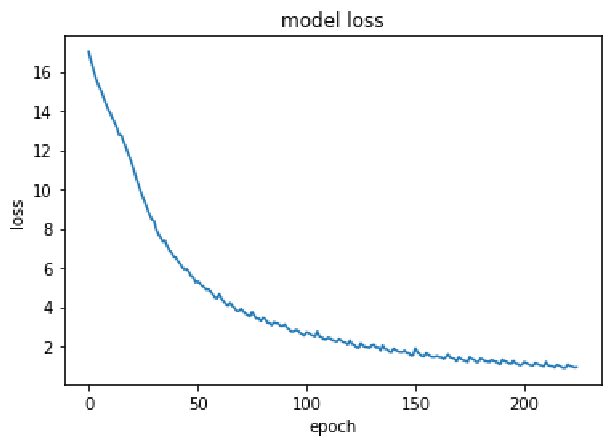


Figure 17: Training loss.

The edit distance (Figure 18) was recorded every 15 epochs on the test set, which is composed of data-points unseen by the model. After the 12th recording, it appears that this metric no longer decreases, which indicates that the model began overfitting.

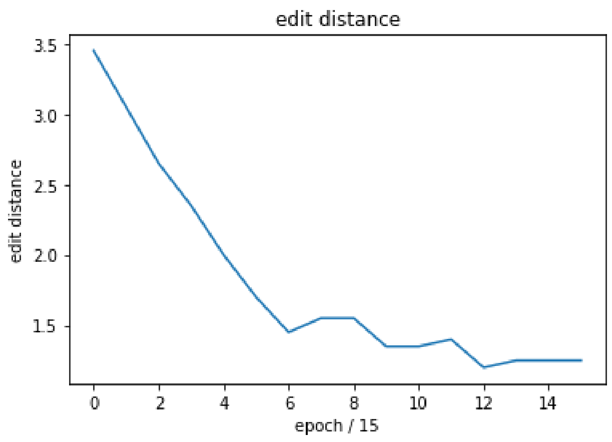


Figure 18: Edit distance of the test set.

4.7 Results

The model achieves a test set edit distance of about 1.2. This means that on average, the model makes one mistake per image, and sometimes two. Although this result is clearly not yet ideal, it shows the potential of these architectures.

The key factor keeping us from obtaining better results is the number of training samples. There is more than enough evidence in the field of deep learning that data is the main ingredient for success [9]. Once the model is unable to continue scaling with more data, increasing the number and size of the layers is usually enough to start experiencing improvements again. It is very safe to say that this model has an incredible room for improvement as more exams are gathered year after year. This is why, in this project, support for easily creating and expanding the training dataset was implemented. More on this in Section 5.3.

5 System Implementation

A simple Python notebook with the code necessary to run a neural network model is far from enough to become useful for any final user. Thus, on top of exploring the different deep learning frameworks that are available and which one was chosen for this project, this section also gives details on the front-end for the user to interact with the model, and a back-end that manages the requests of the user.

Furthermore, since both the Tesseract box prediction tool and the OCR model have frequent inaccuracies, the interface offers comfortable editing of the results of both models, and finally allows the user to save the final predictions to disk for posterior training of the OCR model on the newly generated data.

5.1 Deep Learning Frameworks

An important decision to make when working on a deep learning project is choosing a framework to work with. Although properly evaluating the popularity and usage of software libraries is not an easy task, several surveys have attempted to judge which are the most popular deep learning frameworks out there. According to an analysis from KDNuggets [1], the most popular one is Tensorflow, followed by Keras and PyTorch (see Figure 19). Many more exist, and the field is very active, so larger popularity shifts are expected throughout the years as well.

Tensorflow

Tensorflow is a general purpose, differential programming symbolic math library. Although its main purpose was to accelerate deep learning research and production at Google, they soon open-sourced it and it became one of the first and most popular options in the discipline.

In order to create a neural network with TensorFlow, one needs to define a computational graph through the use of tensor operations. Using these

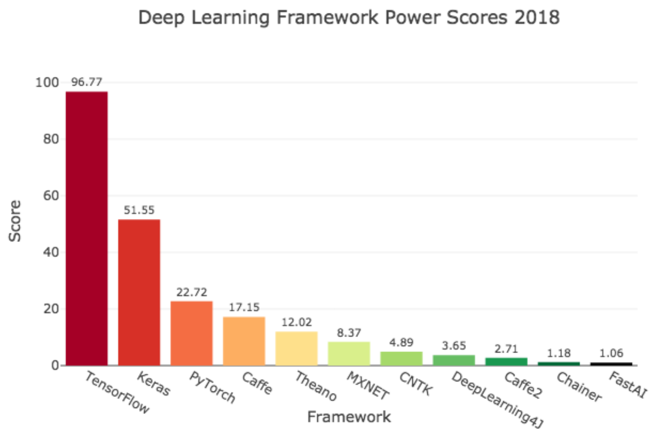


Figure 19: The most popular deep learning frameworks, by [1].

operations in the code does not mean however that they are executed at that line, following a deferred execution principle. In order to execute the defined operations, one needs to 'run' the graph that was previously built.

Its sometimes demanding and complex syntax, as well as its unusual programming paradigm have led to the surge of other easier to use alternatives such as Keras.

Keras

Keras is an open source high level deep learning framework for Python. Essentially, it works as a wrapper to low level libraries, focusing on being user-friendly and modular. It is capable of using four different back-end kernels: TensorFlow, Microsoft Cognitive Toolkit, Theano, or MXNet. It is the preferred choice for many deep learning practitioners due to its ease of use, as well as extensibility and compactness of the code. Its primary author and maintainer is François Chollet. This is the framework that was finally chosen to implement the neural network of this project.

PyTorch

The most popular low level framework alternative for Tensorflow is the Facebook backed PyTorch, with similar functionality but a more imperative approach. Instead of creating computational graphs like is the case by default in Tensorflow, PyTorch executes the operations on the go.

5.2 Using GPUs

Many Keras layers run on the GPU by default if available. However, RNNs are a bit special. They require special implementations in order to run, and thus Keras provides two separate versions of each RNN layer, depending if you wish to run it on a GPU or not. Luckily, if you tweak the parameters of these layers, it is possible to find a configuration that allows for compatibility between the two versions. This means that you can train the model using the GPU enhanced version, and later deploy the model in the back-end using the CPU version, eliminating the requirement of having a GPU available in the target environment.

5.3 Front-end: Editing Boxes and Predictions

A diagram describing the different modules and their interconnection is displayed in Figure 20.

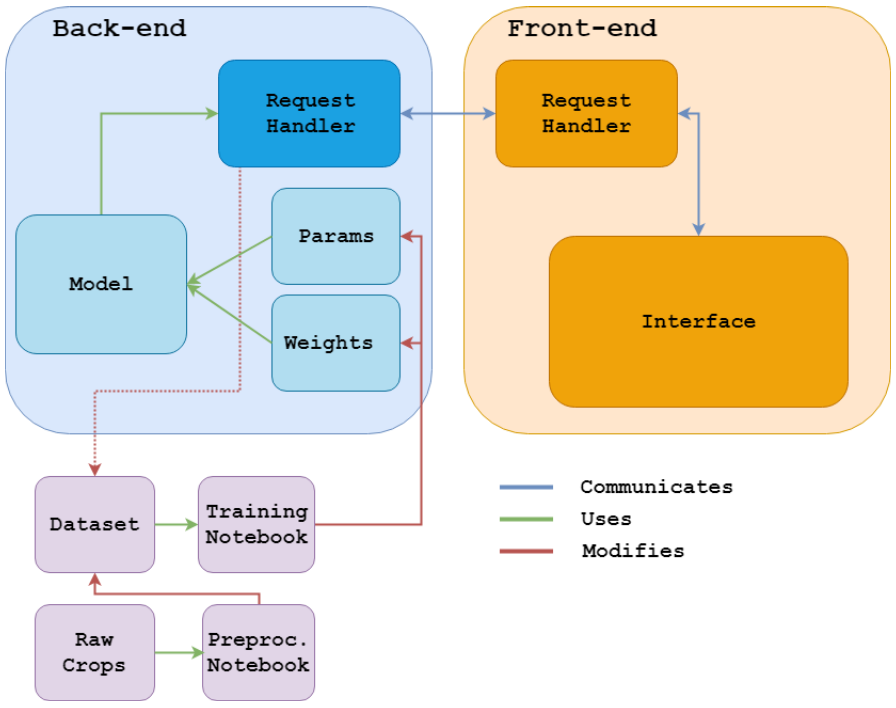


Figure 20: Diagram of the system.

The front-end is implemented using web technologies inside the folder `client`. The HTML file `index.html` offers the basic skeleton. Inside `style.css` are the style definitions for the different components used. Finally, the file `script.js` contains the JavaScript code that powers the interface. The interface is meant to be ran locally as a file, but can easily be extended to work remotely as well, if needed.

In Figure 20, it is represented in orange. To communicate with the back-end, HTTP requests are sent from the browser to the back-end server. This has been conceptualized as a module called *request handler* in the diagram.

The first step for the user is to place all cropped images that he or she desires to work with inside the folder `evaluate` of the project folder. Then, inside the interface, a file can be chosen and uploaded. The image will then appear to the left side of the screen, and will be adjusted to fit both horizontally and vertically inside its assigned space. The scale factor used to adjust the image is saved, since it has to be taken into account when later working with pixel coordinates of the image.

The next step is to request line box coordinates. The back-end is in charge of running the `tesseract` command to obtain them, and pass the values over. The corresponding coordinates in the browser window need to be calculated by using offsets and scaling parameters.

Now the user can resize, move and delete boxes until he or she is satisfied with the result. To allow for the boxes to move, a small gray header is present in the top left corner of them. When the cursor is hovering over these headers, clicking and dragging them will move the whole box. Double clicking the header will remove the box.

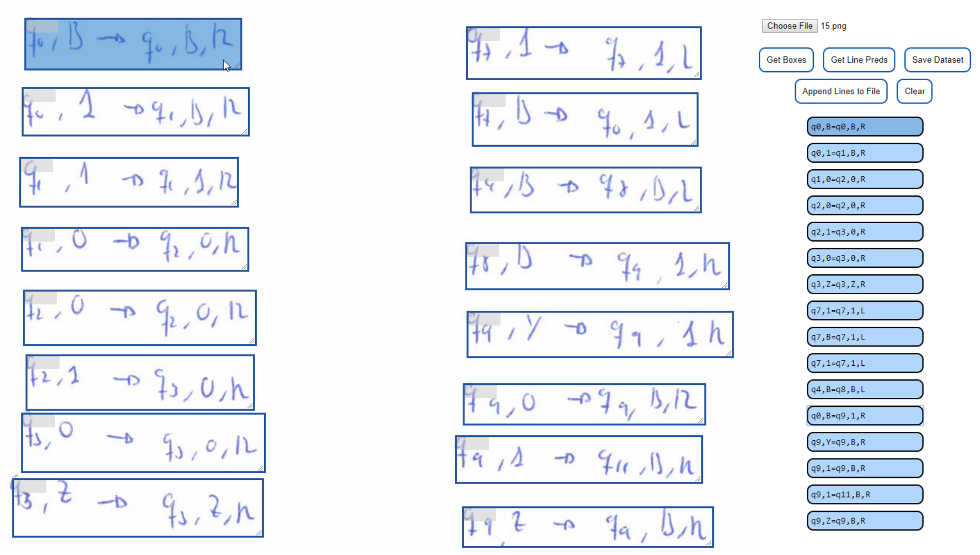
Once the user has adjusted the boxes, their new coordinates can be submitted for OCR evaluation. Upon submission to the back-end, the returned predictions are displayed on the right side of the screen. Once this step is done, it will look similar to what can be seen in Figure 21. When hovering over a box or over a line, the corresponding box/line is also highlighted, allowing the user to easily distinguish which prediction corresponds to each box and vice-versa.

Finally, the user can edit the predictions to fix any mistakes that the OCR has performed, and append the lines to either an existing or a new text file. The intention is that, since a single Turing machine could be composed of several crops, it is sometimes necessary to allow the user to extend an existing file.

Additionally, the user can choose to save the data to a dataset for posterior improvement of the model.

5.4 Back-end: Serving the Model

In order to allow the front-end to communicate with the model, it is embedded in a Flask server application. Flask is a minimalist web framework for Python. The main functionality that was sought after when deciding for a tool to use is easy HTTP request handling. With Flask, only a few



```
q0,B=q0,B,R
q0,1=q1,B,R
q1,0=q2,0,R
q2,0=q2,0,R
q2,1=q3,0,R
q3,0=q3,0,R
q3,Z=q3,Z,R
q7,1=q7,1,L
q7,B=q7,1,L
q7,1=q7,1,L
q4,B=q8,B,L
q0,B=q9,1,R
q9,Y=q9,B,R
q9,1=q9,B,R
q9,1=q11,B,R
q9,Z=q9,B,R
```

Figure 21: State of the interface when user requested OCR lines.

imports and a few function decorators are needed to correctly receive and reply to a request.

The back-end is written in the file `app.py` in the root directory, and can be launched through the terminal command `flask run`. All files related to the model, which will be loaded by the back-end, are contained in the `model` folder. Specifically, these are the model specification as a Python function, the weights, the parameters and configuration, and set of helper functions.

In the diagram, the back-end is represented in blue. Requests of the front-end are captured through the corresponding module and the *model* is queried to obtain the replies that have to be sent back. The model is loaded using the *weights* and *parameters*.

5.5 Notebooks and Tools

Apart from the front- and back-end described above, the project contains a few notebooks, as well as the data, that are used during training. They are represented in purple. The *raw crops* are passed through the *preprocessing* notebook to produce the final *dataset*, and are then used by the *training notebook* to produce the above mentioned weights and parameters necessary for the model. Finally, a dotted red line indicates that the front-end is capable of sending a request to directly modify the dataset by storing additional data samples in it.

For the box generation procedure, the tool Tesseract has ben used. It can be installed following the instructions in their GitHub Wiki page (`https://github.com/tesseract-ocr/tesseract/wiki`). It is a command line tool with several OCR capabilities, but the functionality that we are looking for bounding box detection. This can be obtained by passing the parameter `hocr` to the terminal command. HOCR is an open standard of data representation for text obtained from OCR techniques, using the HTML syntax. This output will contain boxes and the predictions for those boxes. Of course, these predictions are quite poor when evaluated on hand written Turing machines, so we are only concerned with obtaining the coordinates.

The output of the `tesseract` command line program can be captured for each caption, and saved into a string variable. The HTML structure of the string allows it to be parsed to obtain an object data representation, from which then the data corresponding to the coordinates of the boxes can be obtained and returned as a list. For more details, please visit the file `scripts/box_extraction.py`.

6 Conclusions and Future Work

This thesis has covered the theory and implementation details necessary to build, train and deploy a complex neural network model for handwritten character recognition. Key insights have been gained into correctly defining the network's properties, assisting it during the training process, and embedding it into a user-friendly system for deployment.

Although there is still much room for improvement, the results obtained are promising and prove that modern deep learning architectures are powerful and represent an important step forward in this field.

Future work could involve increasing the size of the dataset progressively, and study its correlation with the model accuracy. Once adding more data shows no signs of progression, bigger architectures can be explored and compared to see if they are able to improve on the original structure without sacrificing generalization.

References

[1] Jeff Hale (at KDNuggets). Deep learning framework power scores 2018. `https://www.kdnuggets.com/2018/09/deep-learning-framework-power-scores-2018.html`. (Accessed on 04/11/2019).

[2] Scott Crossley and Victor Kostyuk. Letting the genie out of the lamp: Using natural language processing tools to predict math performance. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 10318 LNAI:330–342, 2017.

[3] Dan Deng, Haifeng Liu, Xuelong Li, and Deng Cai. Pixellink: Detecting scene text via instance segmentation, 2018.

[4] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. ImageNet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee, 2009.

[5] Peter W Foltz, Darrell Laham, and Thomas K Landauer. Automated essay scoring: Applications to educational technology. In *EdMedia+ Innovate Learning*, pages 939–944. Association for the Advancement of Computing in Education (AACE), 1999.

[6] Ross Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. Rich feature hierarchies for accurate object detection and semantic segmentation. *2014 IEEE Conference on Computer Vision and Pattern Recognition*, Jun 2014.

[7] Awni Hannun. Sequence modeling with CTC. `https://distill.pub/2017/ctc`, 2017.

[8] Tong He, Weilin Huang, Yu Qiao, and Jian Yao. Text-attentional convolutional neural network for scene text detection. *IEEE Transactions on Image Processing*, 25(6):2529–2541, Jun 2016.

[9] Joel Hestness, Sharan Narang, Newsha Ardalani, Gregory Diamos, Heewoo Jun, Hassan Kianinejad, Md Patwary, Mostofa Ali, Yang Yang, and Yanqi Zhou. Deep learning scaling is predictable, empirically. *arXiv preprint arXiv:1712.00409*, 2017.

[10] Jan Hosang, Rodrigo Benenson, Piotr Dollár, and Bernt Schiele. What makes for effective detection proposals? *IEEE transactions on pattern analysis and machine intelligence*, 38(4):814–830, 2015.

[11] Wei Jia, Zhuoyao Zhong, Lei Sun, and Qiang Huo. A CNN-based approach to detecting text from images of whiteboards and handwritten

notes. *2018 16th International Conference on Frontiers in Handwriting Recognition (ICFHR)*, pages 1–6, 2018.

[12] Will Knight. The U.S. military wants its autonomous machines to explain themselves - MIT Technology Review. `https://www.technologyreview.com/s/603795/the-us-military-wants-its-autonomous-machines-to-explain-themselves/`. (Accessed on 04/04/2019).

[13] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. ImageNet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[14] Yann LeCun and Corinna Cortes. MNIST handwritten digit database. http://yann.lecun.com/exdb/mnist/, 2010.

[15] Marvin Minsky and Seymour A Papert. *Perceptrons: An introduction to computational geometry.* MIT press, 2017.

[16] Vinod Nair and Geoffrey E Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th international conference on machine learning (ICML-10)*, pages 807–814, 2010.

[17] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. Understanding the exploding gradient problem. *CoRR, abs/1211.5063*, 2, 2012.

[18] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 779–788, 2016.

[19] Shaoqing Ren, Kaiming He, Ross B. Girshick, and Jian Sun. Faster R-CNN: Towards real-time object detection with region proposal networks. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 39:1137–1149, 2015.

[20] Frank Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958.

[21] Sebastian Ruder. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*, 2016.

[22] Jürgen Schmidhuber. Deep learning. *Encyclopedia of Machine Learning and Data Mining*, pages 1–11, 2016.

[23] Mike Schuster and Kuldip K Paliwal. Bidirectional recurrent neural networks. *IEEE Transactions on Signal Processing*, 45(11):2673–2681, 1997.

[24] Baoguang Shi, Xiang Bai, and Cong Yao. An end-to-end trainable neural network for image-based sequence recognition and its application to scene text recognition. *IEEE transactions on pattern analysis and machine intelligence*, 39(11):2298–2304, 2017.

[25] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15(1):1929–1958, 2014.

[26] Jiawei Su, Danilo Vasconcellos Vargas, and Kouichi Sakurai. One pixel attack for fooling deep neural networks. *IEEE Transactions on Evolutionary Computation*, 2019.

[27] Paul Werbos. Beyond regression: New tools for prediction and analysis in the behavioral sciences. *Ph. D. dissertation, Harvard University*, 1974.