Multi-character Prediction Using Attention

by

Mohmmed Baeenh

A thesis submitted to the
School of Graduate and Postdoctoral Studies in partial
fulfillment of the requirements for the degree of

Master of Science in Computer Science

Faculty of Science
Ontario Tech University
Oshawa, Ontario, Canada
January 2020

Thesis Examination Information

Submitted by: **Mohmmed Baeenh**

Master of Science in **Computer Science**

Thesis Title:
Multi-Character Prediction Using Attention.

An oral defense of this thesis took place on *October 03, 2019* in front of the following examining committee:

Examining Committee:

Chair of Examining Committee	**Dr. Alvaro Quevedo**
Research Supervisor	**Dr. Faisal Qureshi**
Examining Committee Member	**Dr. Ken Pu**
Thesis Examiner	**Dr. Bill Kapralos**, *Ontario Tech University*

The above committee determined that the thesis is acceptable in form and content and that a satisfactory knowledge of the field covered by the thesis was demonstrated by the candidate during an oral examination. A signed copy of the Certificate of Approval is available from the School of Graduate and Postdoctoral Studies.

Abstract

Multi-character Prediction Using Attention

Mohmmed Baeenh

Faculty of Science (Computer Science)

Ontario Tech University

2020

We propose a computational attention approach to localize and classify characters in a sequence in a given image. Our approach combines spatial soft-attention with attention regularization and learns "where-to-look" to carry out the sequence classification task. The image is first passed through a Convolutional Neural Network (CNN) that serves as feature extractor. Then at each Recurrent Neural Network (RNN) time step, the attention mechanism attends to the relevant features sequentially to make predictions. The attention mechanism also includes a start and stop state, which instructs the mechanism to start looking and guides it when to stop (e.g., when the sequence has been exhausted). We demonstrate our approach on two sequence detection tasks—multi-digit classification and CAPTCHA unlocking—using the publicly available Street View House Numbers (SVHN) dataset and a custom CAPTCHA dataset. The experiments confirm our hypothesis that the network learns to attend to relevant features by minimizing the loss between the ground truth attention masks and the predicted attention masks.

Keywords: Computational Attention; Convolutional Neural Networks; Recurrent Neural Networks; Multi-Digit Classification; CAPTCHA

Author's Declaration

I hereby declare that this thesis consists of original work of which I have authored. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners. I authorize the *University of Ontario Institute of Technology* to lend this thesis to other institutions or individuals for the purpose of scholarly research. I further authorize the *University of Ontario Institute of Technology* to reproduce this thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research. I understand that my thesis will be made electronically available to the public.

Mohmmed Baeenh
(author)

Statement Of Contributions

I hereby certify that I am the sole author of this thesis and that no part of this thesis has been published. I have used standard referencing practices to acknowledge ideas, research techniques, or other materials that belong to others. Furthermore, I hereby certify that I am sole source of the creative works and/or inventive knowledge described in this thesis.

Acknowledgments

I would like to thank my parents Fekri Baeenh and Ameenh Al-oufi for their unconditional love and support. Thanks are also owed to my brothers and sisters. I am especially indebted to my older sister, Enas Baeenh, who supported me throughout during my graduate studies.

I would like to thank my advisor Dr. Faisal Qureshi for his endless support and encouragement. I am grateful for his patience and continuous guidance throughout my graduate studies.

I would like to thank my colleague and close friend Tony Joseph for the many intellectually stimulating discussions. Tony was always there to answer my Python or LaTeX questions. I will miss our time together.

In addition, I want to thank my close friends Nizar Al-shrief, Ahmad Mohammed, Abdullah Al-Shehri, Ibrahim Diabate, and Ahmad Alqurashi for making graduate studies fun.

Lastly, I am grateful to the Saudi government and the Arabian Cultural Bureau for their generous funding that made it possible for me to pursue graduate studies in Canada.

Contents

List of Tables

List of Figures

CHAPTER 1

Introduction

Over the last decade, deep learning approaches have made tremendous progress on long standing computer vision problems, including face recognition, object classification, medical imaging, etc. [1–8]. Deep networks have been shown to learn to construct powerful image representations that can be subsequently used for complex computer vision tasks. Within this broader context, in this work, we explore a model for computational attention, an emerging area within deep learning. Computational attention models attempt to mimic mechanisms of *attention* found in biological systems [9, 10]. Even insects exhibit behaviours that suggest that these are capable of attending to the relevant sensory inputs. Attention, it has been argued, is closely tied to perception [11]. Indeed it is the first step in perception. Attention helps brain cope with large volumes of sensory data that flows into it through various senses: vision, audition, touch, taste, and smell. Computational attention models then enable a computer to focus on the "relevant" data when carrying out a task, and this has many advantages:

- It will lead to more efficient data processing regimes, since the computer only

Figure 1.1: Samples from SVHN (top row) and CAPTCHA (bottom row) datasets.

needs to store and process part of the data during subsequent steps.

- It may lead to new mechanisms for anomaly detection, since unexpected events should force attention in a particular manner.
- In time, it will lead to algorithms that exhibit traits such as curiosity, excitement, boredom, or annoyance.
- It can be used to gain insight into the inner workings of a system. For example, it can be used to explain the predictions made by an Artificial Intelligence (AI) system.

In order to study computational attention, we looked at the task of sequence classification. Given an image showing a sequence of digits, the sequence classification task attempts to read out the digits in the correct order. Unlike the digit classification task where the number of classes are fixed *a priori*, the number of classes are unknown (and often very large, possibly infinite) for sequence classification tasks. One way to solve this task is to "attend" to each digit one after the other. This requires a model that learns to localize (and classify) one digit at a time in a sequential manner.

In a typical setup, a Convolutional Neural Network (CNN), often termed *encoder*, constructs image features. These features are subsequently used for the task

at hand. Features at different spatial locations encode information present at different regions of the image. Xu *et al*. [12] propose an attention mechanism that learns to sequentially look at different locations in the image features in order to generate an image caption. Their work learns to attend to different regions using a (task) loss that captures network performance at the task at hand. In this work, we combine their model with attention regularization. We show that it is advantageous to use an attention loss in addition to a task loss when training such networks. For our application, the attention loss measures the error between the predicted bounding boxes and the ground truth bounding boxes for each character in the sequence.

We evaluate the proposed approach on two datasets: 1) The Street View House Numbers (SVHN) dataset and 2) a Completely Automated Public Turing test to tell Computers and Humans Apart (CAPTCHA) dataset that we constructed (see Figure 1.1). Previously, Goodfellow *et al*. [13] studied the problem of digit sequence prediction using SVHN dataset. Their approach assumes that the length of the sequence is known *a priori*. Specifically, they propose a deep network with N prediction branches. The first branch predicts the number of digits found in the sequence, and the subsequent branches predict the digits themselves. Their network can predict sequences of length less than N. In their set up N is a hyper-parameter, and the network needs to be re-trained to deal with sequences longer than $N-1$.

Ba *et al*. [14] have also studied the digit sequence classification problem using SVHN dataset. They proposed a deep Recurrent Neural Network (RNN), where at each step, the network processes a multi-resolution crop of the input image, called a glimpse. The network then uses the information from the glimpse to update its internal representation and outputs the next glimpse location and possibly the next object in the sequence. The process continues until the model decides that there are no more objects to process [14]. This approach is trained using the REINFORCE [15] algorithm.

Our method too uses an RNN to sequentially attend to characters in a sequence. Specifically, at each RNN step, soft attention mechanism proposed in [12] is used to focus on a single character in the sequence. Since the soft attention mechanism is differentiable, our model is end-to-end trainable, meaning we are able to train our model using standard gradient descent techniques. Our model also includes a start and a stop state. The start state sets up the sequence reading task; whereas, the stop state indicates that there are no more characters left to be read. The use of start/stop states suggests that our method can classify sequences of any length.[1]

1.1 Contributions

This work makes the following contributions.

I. We present a new end-to-end trainable network for sequence classification using attention mechanism. Unlike existing approaches [12–14, 16–18], we propose to use *attention loss* in addition to *task loss* to train the network. We expect that the proposed model can deal with sequences of any length.

II. We evaluate our approach on SVHN, a standard benchmark for multi-digit classification tasks.

III. In addition, we created a new CAPTCHA dataset, and we evaluated the proposed method on this dataset.

1.2 Thesis Outline

The rest of the thesis is organized as follows. Chapter 2 briefly discusses deep learning background. The following chapter discusses related work. Chapter 4 includes

[1] Due to time constraints, we did not manage to study this in depth, but we plan to apply our approach on longer sequences in the future.

details about the proposed network architecture. An overview of the datasets is provided in Chapter 5. Finally, we end the thesis with a summary and a discussion of possible directions for future work.

CHAPTER 2

Background

This chapter provides a brief introduction to deep learing. For a more in depth treatment of deep learning, we refer the reader to the excellent book on deep learning by Goodfellow *et al*. [19].

2.1 Machine Learning

Machine Learning (ML) is a field in computer science that encompasses algorithms that learn from data. Broadly speaking, we can divide ML algorithms into three categories: 1) supervised learning; 2) unsupervised learning; and 3) reinforcement learning [19,20]. This work deals with supervised learning and we will restrict the rest of the discussion to this branch of ML. Supervised learning deals with learning a mapping from inputs to outputs based on input-output examples. Consequently, supervised learning requires access to labelled data.

2.1.1 Supervised Learning

The goal of supervised learning is to learn a model $y = f(\mathbf{x}; \theta)$ given a set of samples $\{\mathbf{x}_1, \mathbf{x}_2, \cdots, \mathbf{x}_N\}$ and their associated labels $\{y_1, y_2, \cdots, y_N\}$. If y takes on discrete values, we refer to the problem as a *classification* problem. When y takes on continuous values, we refer to this problem as a *regression* problem. Here θ refers to model parameters. The goal of learning is then to estimate parameter values that minimize the error between the predicted values and the ground truth.

Consider, for example, the problem of 2D line fitting. We are given a set of points $\{(x_1, y_1), (x_2, y_2), \cdots, (x_N, y_n)\}$ and our goal is to fit a line $y = \theta_1 + \theta_2 x$ to this set of points. In this case, we can estimate the model parameters θ_1 and θ_2 by minimizing $E = \sum_{i=1}^{N}(y_i - \hat{y}_i)^2$ with respect to the model parameters. Here $\hat{y}_i$ is the model prediction for x_i and y_i is the corresponding ground truth value. While in this toy example, it is possible to estimate model parameters that minimize E using a closed-form analytical expression. This does not hold for more complex models, which do not lend themselves to closed form analytical solutions. An iterative scheme for minimizing E with respect model parameters requires that we compute derivatives of E with respect to these model parameters. We will soon see that such an approach works well for models that are many orders of magnitude more complex than our toy example.

2.2 Neural Networks

Neural Networks (NN) are a class of machine learning algorithms that are able to learn powerful mappings from inputs to outputs given a set of input-output pairs. In its simplest form NN comprise of artificial neurons stacked vertically and horizontally. An artificial neuron is a computational unit inspired by our understanding

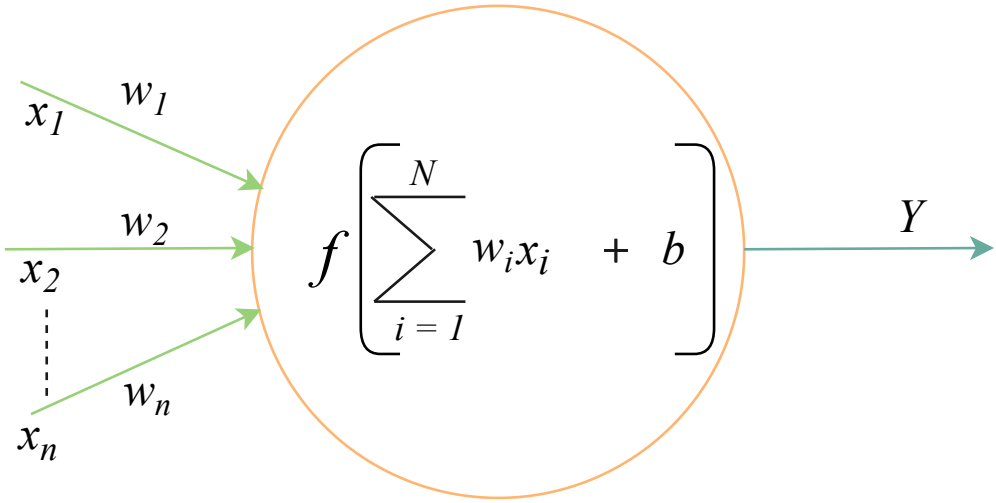


Figure 2.1: An artificial neuron.

of biological neurons. Mathematically,

$$y = f(\mathbf{w}^\top \mathbf{x} + b), \tag{2.1}$$

where $\mathbf{x}$ is the input signal, b is the bias, $\mathbf{w}$ are the connection weights applied to the input signal, f is the activation function, and y is the output of this neuron. See Figure 2.1 for an illustration of the artificial neuron.

2.2.1 Activation Functions

ML researchers have proposed different types of activation functions. We briefly discuss following four activations functions that have seen wide-spread use in NNs (see Figure 2.2): $\mathrm{sigmoid}$, $\tanh$, ReLU, and $\mathrm{LeakyReLU}$. In this work, we use $\mathrm{LeakyReLU}$ activation function.

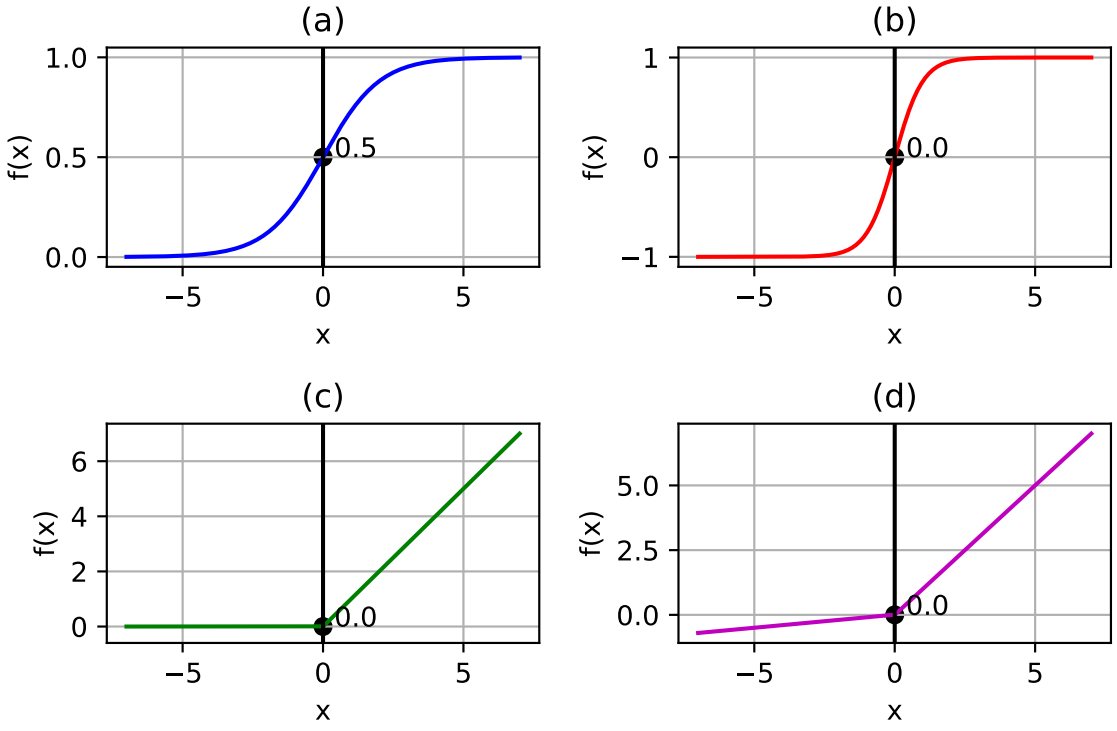


Figure 2.2: Different activation functions used in neural networks. (a) illustrates $sigmoid$ activation function, (b) illustrates $tanh$ activation function, (c) illustrates $ReLU$ activation function, and (d) illustrates $Leaky - ReLU$ activation function. Courtesy ([21])

Sigmoid

A sigmoid is a bounded, differentiable, real-valued function that has a non-negative derivative at each point. It is defined as follows:

$$f(x) = \frac{1}{1 + e^{(-x)}} \tag{2.2}$$

An illustration of sigmoid is shown in Figure 2.2-a

Tanh

This function is defined as the ratio between the hyperbolic sine and the cosine functions (or expanded, as the ratio of the half difference and half sum of two ex-

ponential functions on x and $-x$). It is defined as follows:

$$f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}. \tag{2.3}$$

An illustration of Tanh is shown in Figure 2.2-b

ReLU

This is a unary real valued function, whose graph is shaped like a ramp. It is defined as follows:

$$f(x) = max(0, x). \tag{2.4}$$

An illustration of ReLU is shown in Figure 2.2-c. For activation in regions for $x < 0$, the gradient will be 0 and those neuron weights will not get updated during SGD. This results in dead neurons. This is known as the dying ReLU problem.

Leaky ReLU

Leaky ReLU function has a small negative slope for $x < 0$. It is defined as follows:

$$f(x) = \begin{cases} \alpha x & x \leq 0 \\ x & x > 0, \end{cases} \tag{2.5}$$

where $\alpha \ll 1$ is a hyper-parameter. Leaky-ReLU is an attempt to mitigate the dying ReLU problem by having a small negative slope. An illustration of Leaky ReLU is shown in Figure 2.2-d

2.3 Convolutional Neural Networks

Convolutional neural networks (CNNs) are a subset of neural networks that operate on multi-dimensional data such as images and videos. The first few layers of

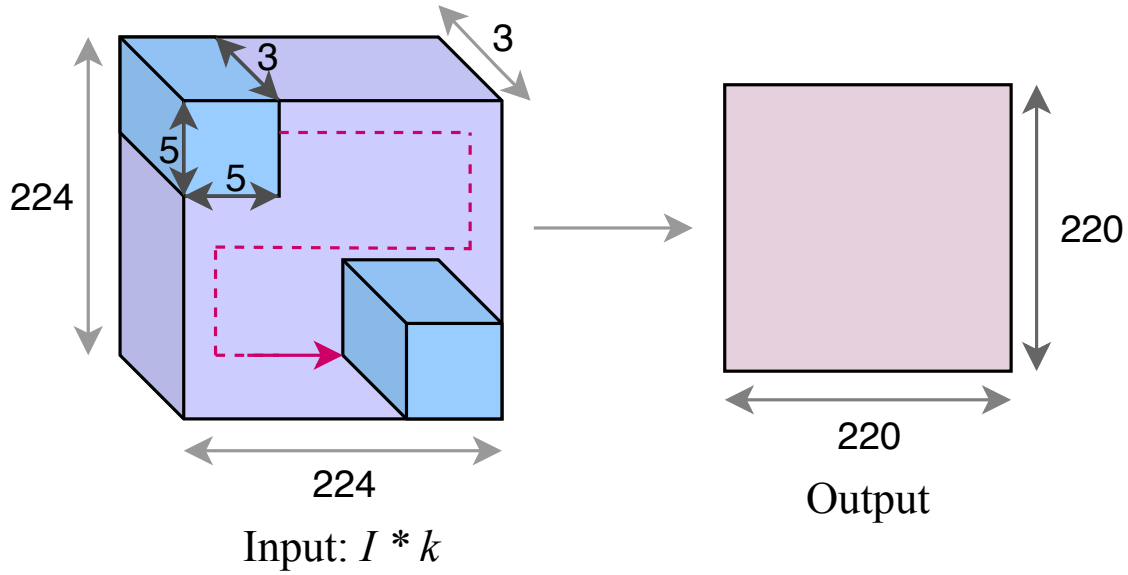


Figure 2.3: Convolution operation on Image $\mathbf{I} \in \mathbb{R}^{224\times224\times3}$ with kernel $\mathbf{k} \in \mathbb{R}^{5\times5\times3}$ (padding $(P) = 0$, stride $(s) = 1$). Based on Equation **??**, the resulting output tensor is $\mathbb{R}^{220\times220}$. (Courtesy [22])

CNNs comprise of convolutional layers, and these are capable of learning powerful image representations that are generally useful at many subsequent computer vision tasks, such as object detection, classification, and recognition. CNNs underpin the recent success of deep learning approaches in a variety of domains [19].

2.3.1 Convolution

In signal processing, convolution is often employed to obtain the overall average of several measurements to get a less noisy signal. Lets consider 1-D example shown below. Given a continuous signal, $\mathbf{x}$, and a filter, $\mathbf{w}$, we define the convolution operator $(*)$ as follows:

$$(\mathbf{x} * \mathbf{w})(t) = \int_{a=-\infty}^{+\infty} \mathbf{x}(t)\mathbf{w}(t-a)dt. \tag{2.6}$$

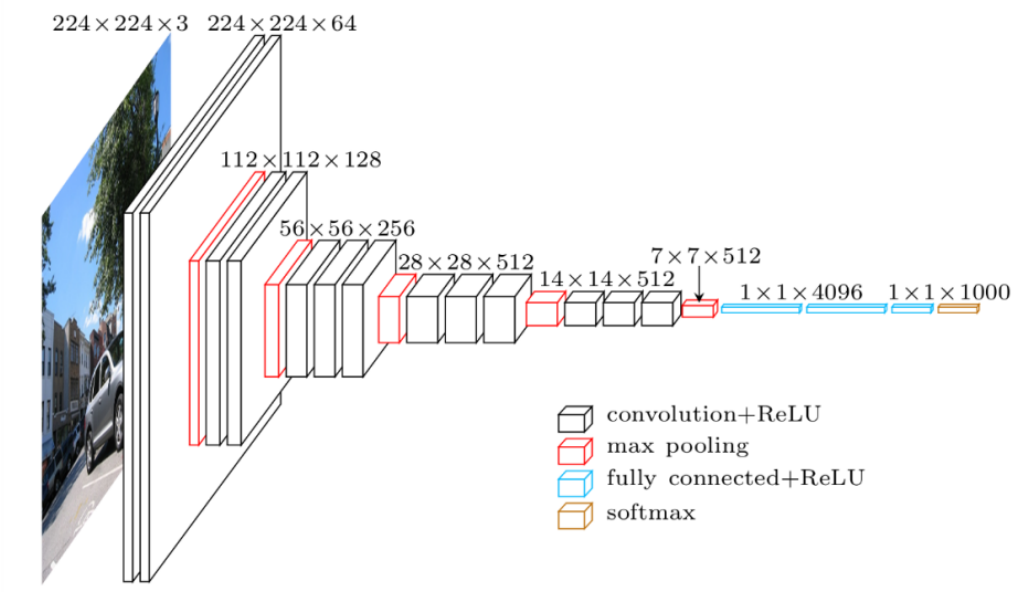


Figure 2.4: VGG-16 convolutional neural networks is a commonly used deep network. It accepts 224×224 RGB images as input. (From [23])

Similarly, for discrete signals, we define convolution as

$$(\mathbf{x} * \mathbf{w})[i] = \sum_m \mathbf{x}[i]\mathbf{w}[i-m]. \tag{2.7}$$

It is easy to extend convolution to more than one dimension. We are interested in images. In the case of a grayscale image, we define 2D convolution operator as follows:

$$(\mathbf{I} * \mathbf{k})[i,j] = \sum_m \sum_n \mathbf{I}[m,n]\mathbf{k}[i-m, j-n]. \tag{2.8}$$

It is worth remembering that deep learning frameworks actually employ *cross-correlation* as opposed to convolutions.

$$(\mathbf{I} * \mathbf{k})[i,j] = \sum_m \sum_n \mathbf{I}[m,n]\mathbf{k}[i+m, j+n]. \tag{2.9}$$

In CNNs, filter (or kernel) parameters are learned from data. If deemed necessary, the network can learn the appropriate values for the flipped kernel that will

result in flipping operation, thereby resulting in a convolutional operation. CNN is made of multiple convolutional layers. Each convolutional layer consists of K filters. Each of these filters is convolved along the input to generate an output. Let the filter height and width be $\mathbf{h}_K$ and width $\mathbf{w}_K$, respectively. Let stride s be amount to shift the filter and the amount of padding P on the borders of the input. The input to the convolutional layer is a feature map with a spatial height $\mathbf{h}_i$, and width $\mathbf{w}_i$. Therefore the spatial output size is given as follows (Figure 2.3 shows an illustration):

$$\mathbf{h}_o = \frac{\mathbf{h}_i - \mathbf{h}_K + 2P}{s} + 1, \text{ and}$$
$$\mathbf{w}_o = \frac{\mathbf{w}_i - \mathbf{w}_K + 2P}{s} + 1.$$

2.3.2 Pooling Layer

The pooling layer is used to spatially reduce the size (downsample) the input feature map. There are two types of pooling: (1) max-pooling and (2) average pooling. Like convolution, the pooling operation is done by sliding a fixed window (kernel) over the input. Max pooling takes the maximum of all the input values inside the kernel. The average pooling takes the average of all the input values inside the kernel. Pooling is performed spatially and independently for each feature map. Figure 2.5 shows an illustration of both pooling mechanisms.

2.4 Batch-Normalization

Batch-Normalization, also known as Batch-Norm, was introduced by Ioffe *et al*. [24] to reduce the training time for deep neural networks. During training, it increases the stability of a neural network by normalizing the outputs of the activations by subtracting the batch mean and dividing by the batch standard deviation. Batch-

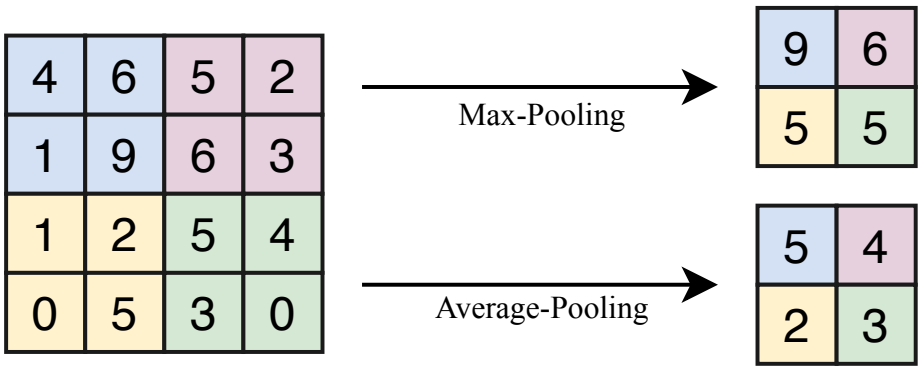


Figure 2.5: Max-pooling and average pooling mechanism. Courtesy ([21])

Norm has two parameters which are trainable if the outputs have to be denormalized. Having these free parameters enables the optimization to change them if it will minimize the loss function. For more information, see to [24].

2.5 Recurrent Neural Networks

Recurrent neural networks (RNNs) are a class of artificial neural networks used for sequence processing. RNNs deal with sequences of inputs vectors and can produce sequences of outputs vectors. The structure of an RNN shown in Figure 2.6. This thesis investigates RNN models for sequential processing and detection, for instance in the case of multi-digit classification using attention, RNN provides a convenient way to transfer relevant information from attention at the present step to the next step. In our case, the input is an image which may consist of multiple digits, and the output can also be sequence numbers predicted from a single image. Therefore, we use RNN to model the sequence processing in this work.

2.5.1 RNN Models for Sequence-to-Sequence prediction

RNNs can deal with input and output sequences of arbitrary lengths, and these have found wide-spread use in sequence-to-sequence translation tasks. Figure 2.6

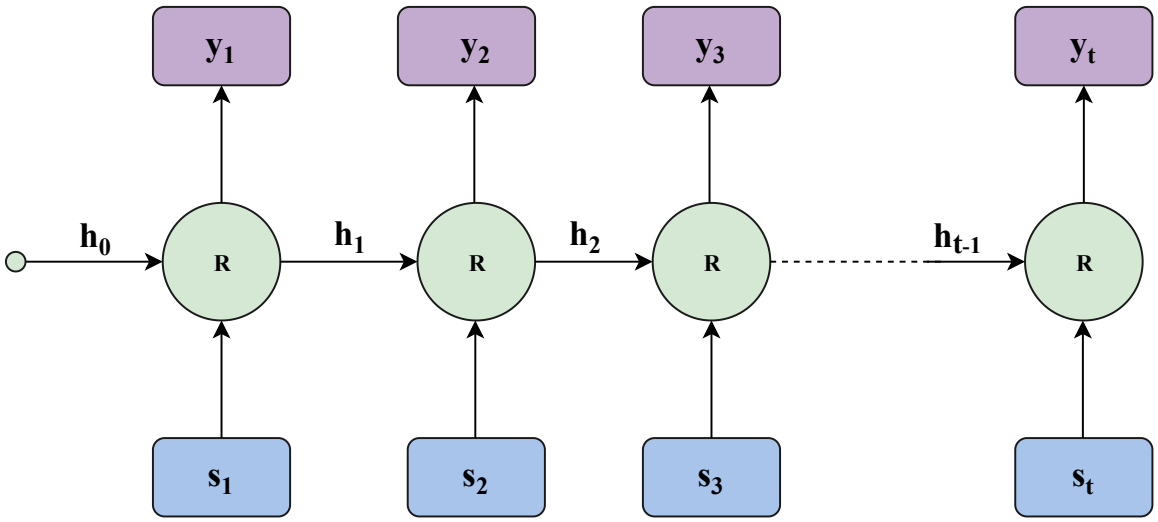


Figure 2.6: Recurrent neural networks. (Courtesy [22])

shows an example of an RNN structure. The input to an RNN is a sequence of vectors $s = (s_1, s_2, ..., s_n)$. At each step in the hidden state is updated as follows:

$$\mathbf{h}_t = \mathcal{R}_{\mathbf{w},b}(\mathbf{h}_{t-1}, \mathbf{s}_t) \tag{2.10}$$

where $\mathcal{R}$ indicates RNN model and $\mathbf{w}$ and b are the shared weights and biases, respectively. Training RNN models is challenging due to the exploding and vanishing gradient problems. Exploding gradient problem is typically solved using gradient clipping and a variation of the RNN model proposed by Hochreiter *et al.* [25] called Long Short-Term Memory Networks (LSTMs) addresses the problem of vanishing gradients.

2.5.2 Long Short-Term Memory Networks

LSTMs consist of a gating mechanism, which enable these to accumulate or forget information conditioned on the task (see Equation 2.11). Specifically, an LSTM consists of three gates: 1) input, 2) forget, and 3) output. In addition to the hidden state, LSTMs also include a cell state. The input and forget gates control cell state

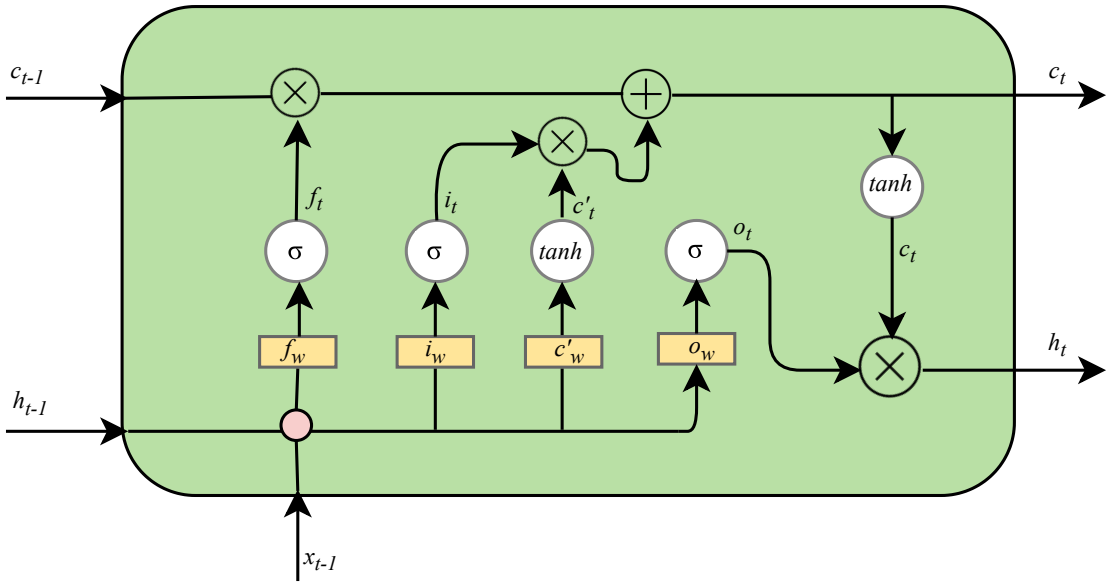


Figure 2.7: Long Short-Term Memory Module. (Courtesy [22])

updates; whereas, the output gate updates the hidden state. If the inputs to the LSTM at each time step are $\mathbf{x}_t \in \mathbb{R}^d$, previous hidden state $\mathbf{h}_{t-1} \in \mathbb{R}^h$, and previous cell state $\mathbf{c}_{t-1} \in \mathbb{R}^h$, then LSTM is implemented as follows:

$$
\begin{aligned}
\mathbf{f}_t &= \sigma_g(\mathbf{W}_f \mathbf{x}_t + \mathbf{U}_f \mathbf{h}_{t-1} + \mathbf{b}_f), \\
\mathbf{i}_t &= \sigma_g(\mathbf{W}_i \mathbf{x}_t + \mathbf{U}_i \mathbf{h}_{t-1} + \mathbf{b}_i), \\
\mathbf{o}_t &= \sigma_g(\mathbf{W}_o \mathbf{x}_t + \mathbf{U}_o \mathbf{h}_{t-1} + \mathbf{b}_o), \\
\mathbf{c}_t &= \mathbf{f}_t \circ \mathbf{c}_{t-1} + \mathbf{i}_t \circ \sigma_c(\mathbf{W}_c \mathbf{x}_t + \mathbf{U}_c \mathbf{h}_{t-1} + \mathbf{b}_c), \text{ and} \\
\mathbf{h}_t &= \mathbf{o}_t \circ \sigma_h(\mathbf{c}_t).
\end{aligned}
\tag{2.11}
$$

Here $\circ$ denotes the Hadamard product, $\mathbf{f}_t$ is the forget gate, $\mathbf{i}_t$ is the input gate, $\mathbf{o}_t$ is the output gate, and $\mathbf{c}_t$ and $\mathbf{h}_t$ are the updated cell and hidden states. An illustration of the LSTM is shown in Figure 2.7.

2.6 Training Neural Networks

In this work, we are concerned with supervised learning. In this setting, we have n data points sampled in an I.I.D. fashion. Each data point i is an $(\mathbf{x}_i, y_i)$ pair, such that $\mathbf{x}_i$ is the input and y_i is the corresponding label (or output). Given a training set, the goal is to learn network parameters (i.e., weights and biases) so as to make a correct prediction y given a previously unseen input $\mathbf{x}$. This is achieved by minimizing some loss function that captures the deviations between network predictions and the ground truth. We discuss the loss functions used in this work in Section 4.5.

2.6.1 Optimization

Neural networks can consist of millions of parameters and directly computing the gradient of the loss with respect to these parameters is generally infeasible. Instead, network parameters are updated using a method called *backpropagation*, which computes the gradients of the loss function with respect to network weights [26].

Given gradient of the loss function with respect to network parameters (or weights), it is possible to update the weights in a manner that would decrease the loss function. The *gradient descent* method does just that. It uses the first order gradients to update the weights in order to minimize the loss function. Modern datasets are often too large, and it is not possible to compute the gradient of the loss function over the entire dataset. This has led stochastic gradient descent (SGD) methods, where gradient of the loss with respect to the network parameters is computed over a small subset (or batch) of the entire training data [27]. Network parameters are updated independently (one after the other) for each batch. In practice SGD is shown to work really well. Batches are shuffled and randomly selected between each epoch. Epoch here refers to a single iteration of the network parameters update, such that the entire training dataset has been shown to the network.

In this work, we use Adam for optimizing network weights. Adam, proposed in [28], is a variant of SGD. Adam uses the following relationship to update network weights given the gradient of the loss with respect to these weights.

$$\begin{aligned}
\widetilde{\mathbf{w}} &= \mathbf{w} + \beta_1 v_{t-1} \\
g &= \nabla_{\mathbf{w}} L(\widetilde{\mathbf{w}}; \hat{y}_i, y_i) \\
r_t &= \beta_2 r_{t-1} + (1-\beta_2) g \odot g \\
v_t &= \beta_1 v_{t-1} - \frac{\alpha}{\sqrt{r_t}} \odot g \\
\mathbf{w} &= \mathbf{w} + v_t.
\end{aligned} \tag{2.12}$$

Here, α is the learning rate which controls the size of the step taken in the direction of the negative gradient. Also, in practice learning rate is decayed after a fixed number of epochs to prevent over-shooting in low-error regions.

As suggested in the literature, we randomly initialize the network weights before training. Specifically, we used Xavier initialization [29], which has shown to help networks converge faster. Neural networks often have thousands (and if not millions) of parameters. Consequently, neural networks are heavily over-parameterized models, and these tend to overfit, i.e., the neural networks will achieve near perfect scores on test data, but do poorly on previously unseen test data. Common techniques to avoid overfitting are *regularization* and *dropout*. In this work, we use L2 regularization [30] and dropout [31] to prevent overfitting.

2.6.2 Training Procedure

The training procedure is described below.

Algorithm 1 Neural network Training Procedure

Require:
  Define the neural network architecture, along with the loss function.
  Select an SGD optimizer i.e. Adam and define it's hyperparameters:
    learning rate, decay rate and momentum coefficients.
  Generate the forward propogation computational graph.
  Generate the gradient computational graph using backprop.
  Initialize network parameters $\mathbf{w}$ using some intialization scheme.
  Initialize training parameters such as: epochs, batch size n, error margin ϵ.
Begin:
  for epoch in epochs
    Initialize accumlation of epoch loss variable, **alv** = 0
    for batch in batches
      Sample n data pairs from training set.
      Obtain predictions $\hat{y_n}$ through forward propogation of input x_n.
      Compute gradients, $\nabla_{\mathbf{w}} J(\mathbf{w}; \hat{y_n}, y_n)$, using backprop.
      Compute and update the SGD coefficients.
      Use the updated SGD coefficients to compute gradient direction $\Delta\mathbf{w}$.
      Apply update to the parameters: $\mathbf{w} = \mathbf{w} + \Delta\mathbf{w}$.
      Increment **alv**: alv $+L(\hat{y_n}, y_n)$
    end
    if (**alv** $\div$ total training samples) $< \epsilon$ **then**
      break
  end
end

2.7 Summary

In this chapter, we have identified and discussed the relevant machinery for deep neural networks. We now turn our attention to the problem at hand.

CHAPTER 3

Related Works

This chapter briefly discusses related work in attention and sequence prediction. This discussion is by no means complete. Our goal is simply to highlight the novel aspects of the work presented here.

3.1 Feature Extraction

In computer vision, we don't directly work with raw pixels, but rather extract some useful representations from images termed *features*, which are then used for further processing. In this work, we use Convolutional Neural Networks (CNNs) as our feature extractor. In this section, we motivate the use of CNNs as our feature extractor.

Hand-crafted techniques for constructing image features predate deep learning approaches. These older techniques often rely upon local information present in the image, such as edges, blobs, color, etc., to construct features that are useful for a variety of computer vision tasks. Examples of hand-crafted techniques for image

feature construction are: Scale-Invariant Feature Transform (SIFT) [32], Speeded-Up Robust Features (SURF) [33], and Binary Robust Independent Elementary Features (BRIEF) [34].

Convolutional Neural Networks (CNN) have found unprecedented success at many computer vision tasks, such as object detection, semantic segmentation, video tracking, etc. [13, 35–38]. Given a training set, CNNs are able to learn how to construct powerful image representations (or features). To a large extent, CNNs have replaced classical hand-crafted image feature construction techniques.

Previous works such as [39], and its successors YOLO [5], Fast/Faster/Mask R-CNN [6, 7, 40] have been shown to detect and segment objects in an image using a single feed-forward pass. These methods, first, obtain features using a CNN and then uses two techniques: (1) region proposals from features using RPN (Region Proposal Network) and (2) Region of Interest (RoI) pooling to reshape the relevant proposals into a fixed size. Next, the reshaped features are passed onto another fully connected layer to predict the class of the proposed region as well to regress the bounding box coordinates for the region.

In this work, we too use a CNN to construct image features. However, instead of region proposals, we directly attend to the relevant features to localize (detect) and classify digits in a sequential manner.

3.2 Attention

Attention has been gaining significant research interest recently for various machine learning and computer vision tasks [12, 14, 16–18]. Attention mechanisms learn to probe the input spatially in a sequential fashion, attending to relevant regions to make predictions. There are two types of attention mechanisms: (1) soft attention and (2) hard attention. In practice, these mechanisms are learnable neu-

ral networks which are embedded into the overall network. Soft attention assigns weights to the inputs (features), giving a higher weight to relevant regions and lower weights to non-relevant regions. Whereas, hard attention involves selecting a single region or a feature which is deemed most important. The soft attention mechanism is differentiable with respect to its parameters allowing gradients to be computed using backpropagation. This allows for an end-to-end trainable network. The hard attention mechanism, on the other hand, involves a discrete selection, which is not differentiable. In this case, the network is trained using reinforcement learning techniques, such as REINFORCE [15]. Recently, reparameterization tricks [41, 42] are used to enable backpropagation through the hard selection mechanism. This makes it is possible to achieve end-to-end training for hard attention. In this work, we use soft-attention mechanism (from now on referred to as attention) for digit and character localization.

3.2.1 Sequential Attention Models

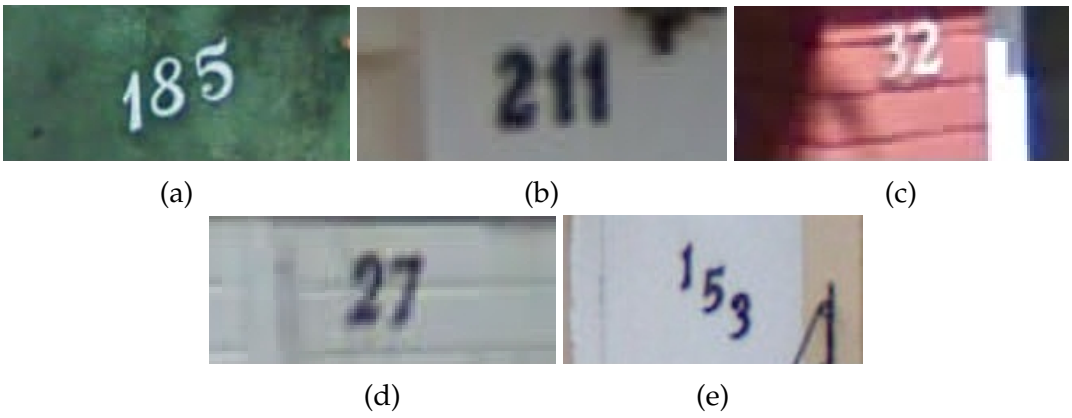
An illustration of the image captioning using attention is shown in Figure 3.1. The attention module needs to attend to different regions in the image in a sequential manner to construct an appropriate caption for the image. Such sequential decision making process are modelled using a class of neural networks called Recurrent Neural Networks (RNNs) [43–45]. In this work, we use an LSTM [25] network to attend to different regions in the image in order to perform digit sequence classification and to find CAPTCHA solution.

Ba *et al*. [14] investigates an attention based model, which is able to localize and predict multiple objects in an image sequentially using RNN. Their attention model is termed the *glimpse* network. If there are multiple objects in the input image, each object gets a fixed number of glimpses (each glimpse is a recurrent step) before a prediction is made. Suppose there are five objects in an image, and three glimpses

Figure 3.1: Illustration of the soft attention mechanism applied to image captioning. *White* regions roughly indicates where the model attends spatially. (Courtesy [12])

per object, a total of 15 glimpses/recurrent steps are needed. Their model uses hard selection and is trained using the REINFORCE method. This work is similar to our work, except in our case, we use a soft attention mechanism. In addition, our system localize and predict at each recurrent step. Therefore, only a total of seven (start-state + five steps + stop-state) recurrent steps are needed.

Meng *et al*. [46] utilized an attention mechanism for action recognition in video. They proposed an attention mechanism termed *Where* and *When* to look. Their attention mechanism attends to the most important parts in the video, and they apply soft attention. To obtain a better performance in their models, they present various regularizers in their models which are spatial attention and temporal attention.



(a) (b) (c)

(d) (e)

Figure 3.2: Sample images of SVHN dataset.

3.3 Multi-Digit classification

For multi-digit classification task, we want to identify multiple digits appearing as a sequence in a single image. Ian *et al*. [13] proposed a model for multiple digit recognition in a single image. Their model consists of feature extractor and fully connected layers. The feature extractor is a CNN consisting of eight convolutional hidden layers. It is followed by two fully connected hidden layers, which consists of $3,072$ neurons each. Finally, the model consists of six prediction branches. The first branch predicts the sequence length and the other five, the digits (the evaluation dataset SVHN [47] has a maximum of five digits in an image). The major limitation of this approach is that the maximum digits it can recognize is fixed (in this case five). If more digits are needed to be recognised then a new model has to be re-trained with more prediction branches added to account for the extra digits. In our work, we have a start and stop state that enables our model to sequentially keep detecting digits (beyond the maximum recurrent steps our model was trained for), until the stop-state is reached. Ba *et al*. [14] and Mnih *et al*. [48] proposed a similar model for a multiple object recognition in a single input image. At each time step the model probes through the input image to effectively classify multiple objects. Especially, Ba *et al*. [14] trained a model to classify all the digits in an image sequentially.

Figure 3.3: Samples of CAPTCHA images. This is used for providing a visual turing test. This dataset is not publicaly available, therefore we generated our own CAPTCHA dataset (more details in Section **??**) for conducting experiments. (Courtesy [13])

The label sequence order is chosen to go from left to right as the natural ordering of the house number. Both [13, 14] evaluate their approach using Street View House Number (SVHN) [47]. The highest sequence accuracy, 96.1%, is obtained by [14].

3.4 CAPTCHA

CAPTCHA is a common challenge-response *Turing* test, often used to determine whether or not the user on the other side is human. It consists of an image showing a sequence of letters or numbers distorted to make it difficult for a human to read. It is assumed that CAPTCHA text would be difficult to "read" for an automated optical character recognition algorithm. To evaluate our approach on this task effectively, we trained a baseline model of [13] on our generated CAPTCHA dataset.

3.5 Summary

In this chapter, we briefly discussed the relevant work. Our focus has been on attention based approaches for sequence prediction/classification tasks.

CHAPTER 4

Methodology

We now describe our approach to digit sequence classification. The proposed approach both localizes and classifies the digits (or characters, in case of CAPTCHA) seen in the image. Specifically, at each time step t, the localization module outputs the bounding-box, $\mathbf{b} \in \mathbb{R}^4$ (top-left, top-right, height, and width), and the classification module outputs the associated category, $\mathbf{o} \in \mathbb{R}^C$. In case of multi-digit classifications C is 12 ($0-9$ + start state + stop state).

In Section 4.1, we describe our overall network architecture. Sections 4.3 and 4.4 provide a detailed description of localization and classification modules respectively.

4.1 Network Architecture

We train our model in an end-to-end fashion to detect and classify multiple digit in a sequence seen in a single image. The input to our network is an RGB image, $\mathbf{I} \in \mathbb{R}^{H \times W \times 3}$. The network consists of three sub-networks: (1) Feature extractor, (2)

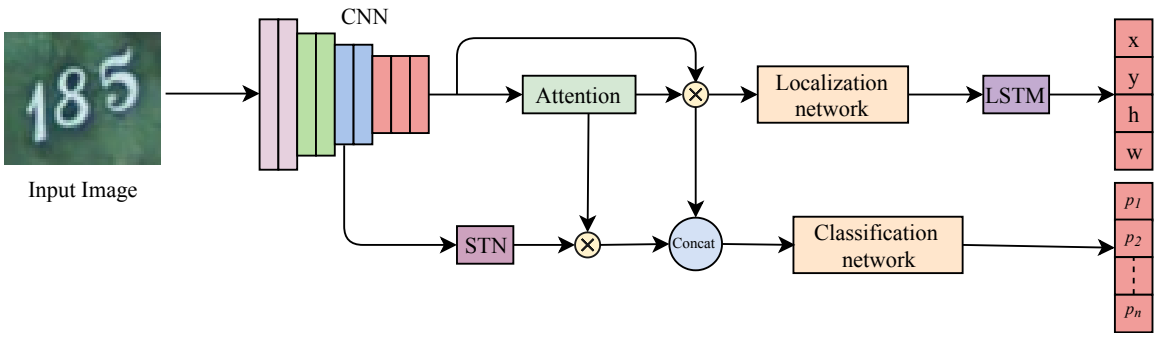


Figure 4.1: Our model takes an RGB image as the input and sequentially predicts the bounding boxes and character predictions until a stop state is reached. The localization network regresses top-left (x), top-right (y), height (h), and width (w) of the bounding box, and the classification network predicts class scores $\mathbf{p} \in \mathbb{R}^C$, where C is the number of classes.

Localization network, and (3) Classification network (see Figure 4.1).

4.1.1 Feature extractor

The feature extractor used in this work is a eight layers convolutional network, which extract features, $\mathbf{f} \in \mathbb{R}^{h_f \times w_f \times 128}$, from the input image, $\mathbf{I}$. These features are then passed onto a localization network.

4.1.2 Localization network

The localization network consists of an attention mechanism and an LSTM network. At each recurrent step, the attention mechanism attends to a feature location corresponding to a single digit/character in the image. The attended features, $\mathbf{f_a} \in \mathbb{R}^{h_{fa} \times w_{fa} \times 128}$, are then used to predict the corresponding bounding box location. In addition, the attended features are used as input to the classification network.

The attention mechanism then jumps to the location corresponding to the adjacent digit (or character). This process continues until no more digits (or characters

are found). Or in other others, until the stop state is reached.

4.1.3 Classification network

The output of the attention mechanism along with the output from the fourth convolutional layer of the CNN is used to predict the category. The fourth layer output, $\mathbf{f_4} \in \mathbb{R}^{h_{f_4} \times w_{f_4} \times 128}$, is passed through a Spatial Transformer Network (STN). $\mathbf{f_4}$ and then average pooled to match the spatial size of $\mathbf{f_a}$. After which both features ($\mathbf{f_{4_{avg}}}, \mathbf{f_a}$) are concatenated along the depth dimension and further average pooled. They are then flattened and fed to two fully-connected layers: $\mathbf{f_{fc1}} \in \mathbb{R}^D$ and $\mathbf{f_{fc2}} \in \mathbb{R}^C$ to make the final prediction

4.2 Feature Extraction

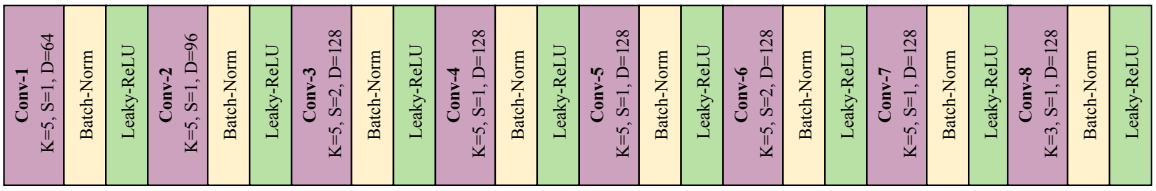


Figure 4.2: This figure illustrates the network architecture. K stands for the kernel size, S stands for the stride, and D stands for the number of kernels in the layer.

We employ a CNN as our feature extractor. It consists of eight convolutional layers. The CNN architecture is shown in Figure 4.2. For each layer, the number of kernels are fixed to 128. The first seven layers use 5×5 kernels. The last layer uses a 3×3 kernel. Batch-norm is added after each convolutional layer, followed by Leaky-ReLU activation function. For reference, we include TensorFlow implementation for our feature extractor in the Appendix B.1.

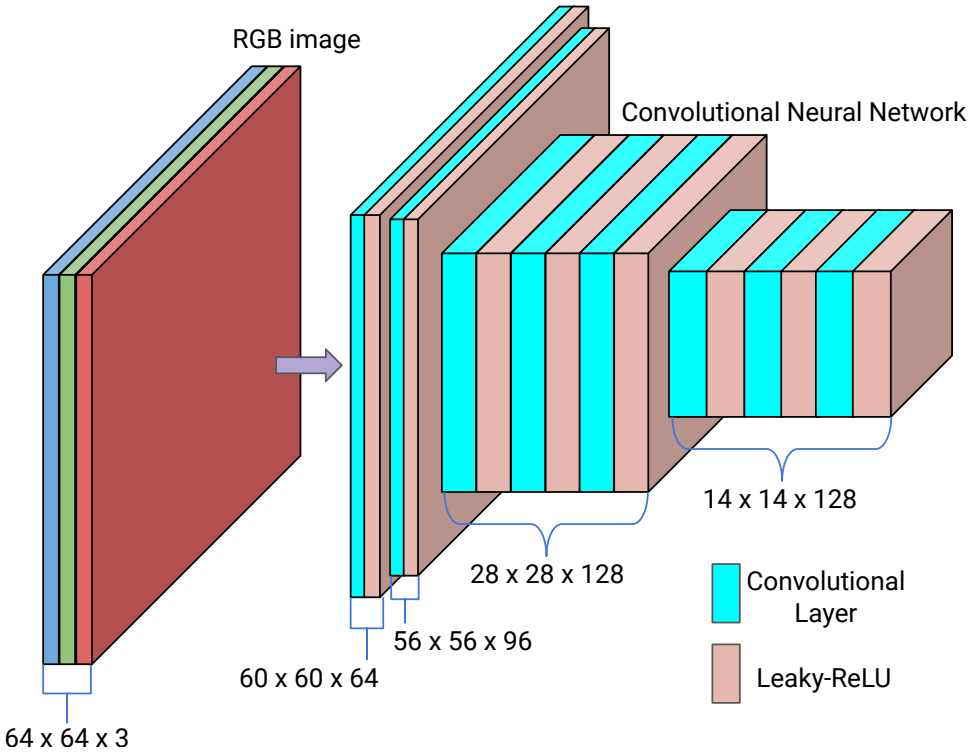


Figure 4.3: This figure shows the output of each CNN layer described in Figure 4.2, when the input is a RGB image of size, $64 \times 64 \times 128$. The final feature output of the CNN in this case is of size, $14 \times 14 \times 128$.

4.3 Localization Module

We use a soft attention mechanism to learn to focus on single digits (characters) in turn. Attention provides a glimpse into the inner workings of the network. The soft attention mechanism is adapted from Xu *et al*. [12]. At each time step t, the input to the attention network is the final features extracted by the CNN along with the previous LSTM hidden state.

4.3.1 Attention Mechanism.

The attention mechanism is implemented as follows:

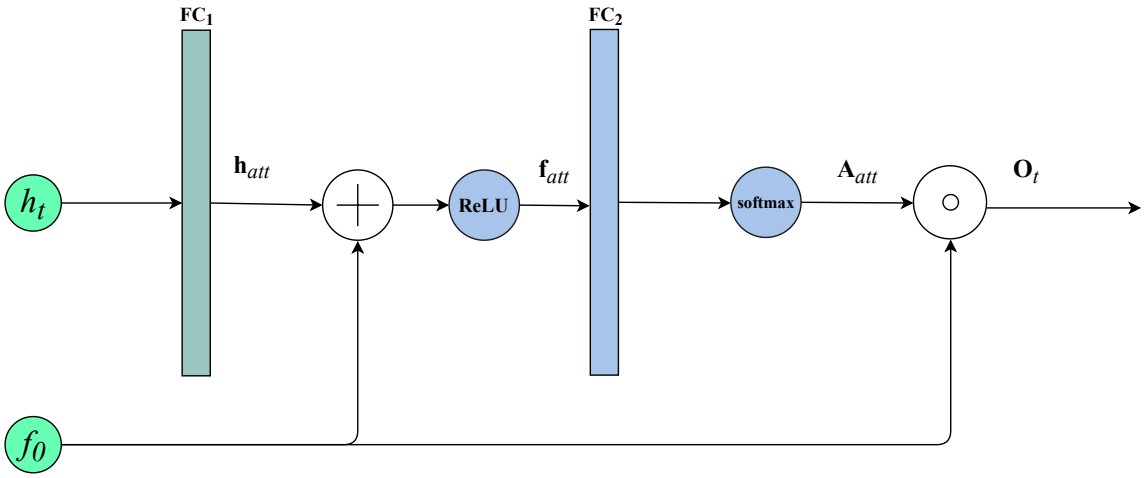


Figure 4.4: Block illustratation of the soft attention model described in Equation 4.1.

$$
\begin{aligned}
\mathbf{h}_{att} &= FC_1(\mathbf{h}_t) \\
\mathbf{f}_{att} &= \mathrm{ReLU}(\mathbf{h}_{att} + \mathbf{f}) \\
\mathbf{A}_{att} &= FC_2(\mathbf{f}_{att}) \\
\mathbf{A}_{att} &= \mathrm{softmax}(\mathbf{A}_{att}) \\
\mathbf{O}_t &= \mathbf{A}_{att} \circ \mathbf{f}.
\end{aligned}
\tag{4.1}
$$

Here, FC_1 and FC_2 are fully-connected neural networks. The attention module consists of two FC layers. The first fully connected layer, FC_1, is used to transform the hidden state $\mathbf{h}_t$, size to bring it equal to the input feature $\mathbf{f}$ size. The second layer, FC_2, computes the unscaled attention mask. The final attention mask is computed by taking the $\mathrm{softmax}$ of the unscaled attention mask. The output of the attention layer $\mathbf{O}_t$ is obtained by taking an element-wise multiplication (Hadamard product) between the input features and attention mask. The TensorFlow implementation of the attention module is shown in code listing B.3.

4.4 Detection Module

At each time step, the attended features are concatenated (along the channel dimension) with the output from STN, averaged pooled, and passed through a fully-connected layer for prediction.

4.4.1 Spatial Transformer Network (STN)

STN was first introduced by Jaderberg *et al*. [49] for models to learn invariance to translation, scale, rotation, and more generic wrappings. STN is a differentiable module which can be inserted into convolutional layers, and it applies a spatial transformation to input feature map. The STN network consists of three components: (1) Localisation network, (2) Grid generator, and (3) Sampler. An illustration of STN is shown in Figure 4.5.

Localisation network

It is a neural network that is part of STN, which outputs the transformation parameters θ. The input to the localization network is a feature map $\mathbf{f}$, and it outputs affine transformation parameters, $\theta \in \mathbb{R}^6$. The input feature map is flattened and passed on to a fully-connected layer in order to regress the parameters. The Tensorflow implementation of the STN localization network is shown in code listing B.2.

Grid generator

The grid generator generates a normalized sampling grid such that it is spatially bounded between $-1 \leq (x_i, y_j) \leq 1$, where $i \in [0, ..., W]$ and $j \in [0, ..., H]$. W is the width, and H is the height, which are the same size as the input feature map size. The normalized grid is then transformed using the transformation matrix whose parameters are obtained from the localization network. This generates

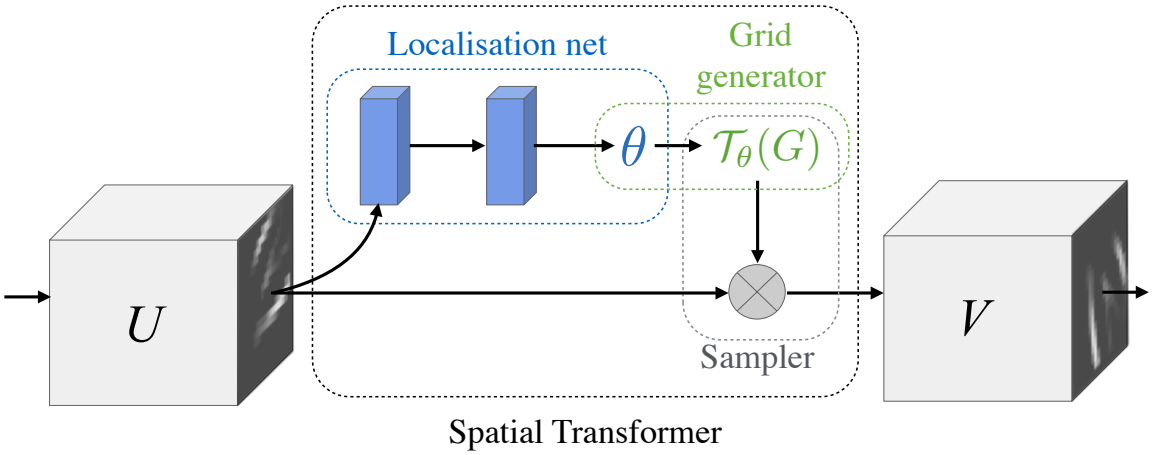


Figure 4.5: Spatial Transformer Network. Figure obtained from [49].

a transformed sampling grid, S. The Tensorflow implementation of the STN grid generator is shown in code listing B.4.

Sampler

It takes the sampling coordinates, S, and samples from an input feature map. The same sampling coordinates, S, are used to perform sampling for each channel of the input feature map. In this work, a bilinear sampling is used. The bilinear sampling is defined as follows:

$$\mathbf{f}' = \sum_{y}^{H} \sum_{x}^{W} \mathbf{f}_{x,y} max(0, 1 - |\mathbf{S}^0_{(x,y)} - x|) max(0, 1 - |\mathbf{S}^1_{(x,y)} - y|), \tag{4.2}$$

where f is the input feature map. This sampling mechanism is (sub-)differentiable, enabling an end-to-end system [49]. The tensorflow implementation of the STN sampler is shown in code listing B.5.

4.5 Loss Functions

Our final loss function is given as follows:

$$\mathcal{L}_{\text{total}} = \mathcal{L}_{\text{bbox}} + \mathcal{L}_{\text{classify}} + \mathcal{L}_{\text{mask}} \tag{4.3}$$

Each of the losses are described below.

Localization loss

$$\mathcal{L}_{\text{bbox}} = \|\mathbf{y}_b - \hat{\mathbf{y}}_b\|_2 \tag{4.4}$$

This is the regression loss—the mean-squared error between the expected value $\mathbf{y}_b$ and the predicted value $\hat{\mathbf{y}}_b$.

Classification loss

$$\mathcal{L}_{\text{classify}} = -\mathbf{y}_c^\top \log(\hat{\mathbf{y}}_c) \tag{4.5}$$

This is the classification loss—the cross-entropy loss between the expected value $\mathbf{y}_c$ and the predicted value $\hat{\mathbf{y}}_c$, which is a one-hot vector.

Attention regularization

This loss penalizes the predicted attention mask as follows:

$$\mathcal{L}_{\text{mask}} = \alpha * log\big(\frac{\alpha}{\hat{\alpha}}\big), \tag{4.6}$$

where α is the ground truth attention mask and $\hat{\alpha}$ is the predicted attention mask. We use Kullback–Leibler (KL) divergence to minimize the information loss between the ground truth and predicted attention masks (both are probability distributions). Recall that KL divergence is a way of measuring the distance between two probability distributions.

4.6 Summary

This chapter discusses the proposed network architecture. We discuss results in the next chapter.

CHAPTER 5

Experimental Results

We now discuss the experimental setup. Our models are implemented in TensorFlow (version 1.4) deep learning framework [50]. All models were trained in an end-to-end fashion using ADAM optimizer [28].

5.1 Datasets

5.1.1 Street View House Numbers Dataset (SVHN)

SVHN is a real-world image dataset for multiple digit recognition [47]. It is similar in flavour to MNIST [51] (e.g., the images contain small cropped digits), but incorporates an order of magnitude more labelled data (over 600,000 images). The data comes from a significantly harder, unsolved, real-world problem (recognizing numbers in natural scene images). SVHN is obtained from house numbers in Google Street View images. We used the standard train/test split for training and inference on SVHN dataset. Figures 5.1 and 5.2 show samples from the training and

test sets, respectively. 10% of the training data was used as a validation set.

The CAPTCHA dataset used by Ian *et al*. in [13] is not publicly available; therefore, we generated our own dataset. We modified the open-sourced CAPTCHA library to generate random CAPTCHA characters of length six.[1] We also store the ground truth bounding boxes for each character. The CAPTCHA characters can consist of both letters and numbers. For generating a single sample, we first sample six characters (letters and numbers) and generate the image. We then apply random rotations to each character. The rotation angle is randomly sampled between -30 and $+30$ degrees. We then paste these images to a blank image canvas from left to right fashion. We also record the (x, y) position where we paste each image along with its corresponding height and width. Thus, we also get the ground truth bounding boxes for each character. Our code for CAPTCHA generation is included in the appendix B.6. Figures 5.3 and 5.4 show training and test samples, respectively, from our dataset. We generated $40,000$ images for training purposes. In addition, we generated $15,000$ images for validation set and another $15,000$ images for test set.

5.1.2 Data-Preprocessing

Street View House Numbers Dataset

We follow the regime used in [13] and [14] to preprocess SVHN dataset. First, a rectangular bounding box is computed for the digit sequence. Next, this box is expanded by 30% along x and y directions, and the image is cropped using this expanded bounding box. The cropped image is resized to 64×64. The ground truth bounding boxes are also re-scaled to match the new image dimensions. We also normalize the pixel intensities between 0 and 1. For training, a batch size of 64 is used.

[1] https://github.com/lepture/captcha

Figure 5.1: Samples from SVHN training set.

Figure 5.2: Samples from SVHN test set.

Figure 5.3: Samples from CAPTCHA training set.

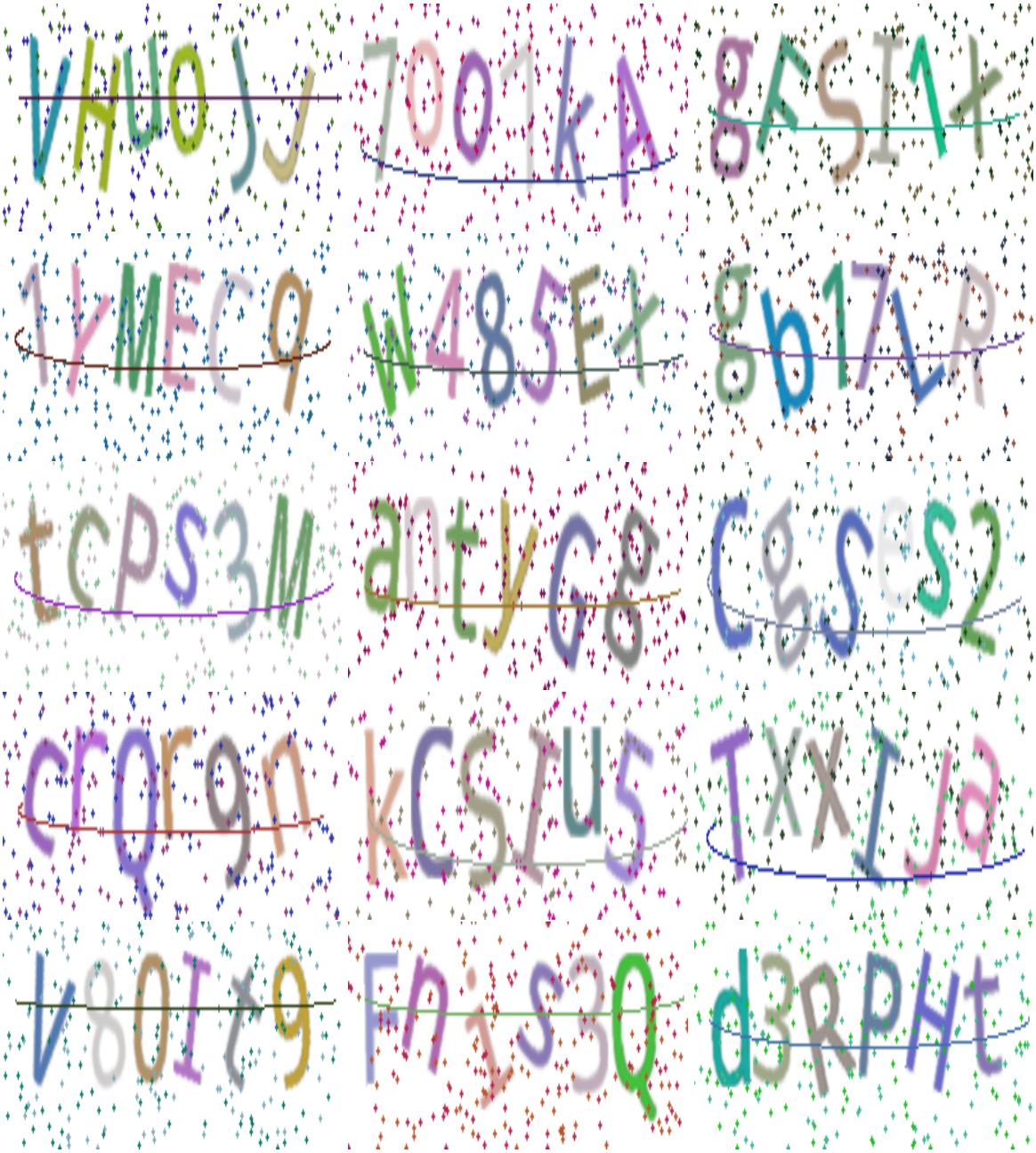


Figure 5.4: Samples from CAPTCHA test set.

CAPTCHA Dataset

For our CAPTCHA dataset, the height of each image is fixed to 64 pixels and the width of each image is fixed to 200. The ground truth bounding boxes are re-scaled to account for the new image dimensions. The pixels intensities are normalized between 0 and 1. For training, a batch size of 64 is used.

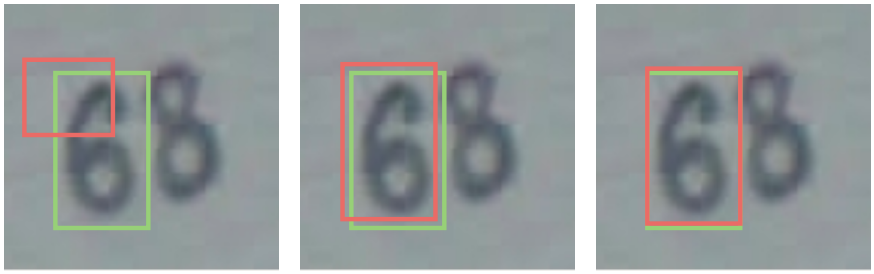
5.2 Discussion

Table 5.2 compares our approach against other baseline methods. The scores show the sequence accuracy results. We compute accuracy as follows. Say the image shows digit sequence: 3, 4, and 5. If the model predicts 3, 4, and 9 then the accuracy is

$$\frac{1+1+0}{3} = 0.67.$$

Table 5.1 shows a comparison between our method and other approaches. The baseline consists of six CNN layers followed by two 1024 fully-connected layers. It assumes that the ground-truth bounding boxes are known, i.e. the network doesn't need to first localize a digit before classifying it. It is important to remember that our network both localizes and classifies the sequence. The high baseline scores suggest that localization accuracy effects classification scores.

The results suggest that the proposed method performs worse than previous approaches by Goodfellow *et al*. [13] and Ba *et al*. [14]. However, one advantage of our approach over the method by Goodfellow *et al*. is that our method can deal with digit sequences of arbitrary length. We make this assertion due to presence of start and stop states. Specifically, unlike their method our method does not need to be re-trained to work with sequences of different maximum length.



(a) Poor (b) Good (c) Excellent

Figure 5.5: Intersection-Over-Union (IOU) computes the area of overlap (intersection) divided by area of union between the ground-truth bounding box (shown in green) and the predicted bounding box (shown in red). IOU takes into account how closely the predicted box and ground-truth boxes match.

Table 5.3 lists Intersection-Over-Union (IOU) scores for our method for the two datasets. IOU score is the overlap area of the two bounding boxes divided by the total area of these bounding boxes. IOU scores capture how closely the predicted bounding box matches the ground truth bounding box (see Figure 5.5).

Figures 5.6 and 5.7 show the qualitative results of our method on SVHN and CAPTCHA datasets, respectively.

5.3 Summary

While our method is unable to match the performance of existing methods, the proposed method does boast the following advantages: 1) the proposed method combines localization and detection to achieve digit sequence prediction; 2) the proposed method does not need to be re-trained to deal with sequences of longer lengths due to the use of start and stop states; and 3) the proposed method can be trained in an end-to-end fashion.

Table 5.1: Comparison between our approach and other existing approaches.

Goodfellow *et al.* [13]	**Ba *et al.*** [14]	**Baseline**	**Ours**
Eight convolutional layers with two fully-connected layers, followed by prediction branches.	Deep,recurrent neural network that at each step process a multi-resolution crop of the input image called a,glimpse.	Six convolutional layers with one fully-connected layer, followed by prediction branch.	Eight convolutional layers followed by attention mechanism.
Direct prediction	Hard attention [Trained using REINFORCE algorithm] with end-state	Input is the character extracted using ground-truth bounding box.	Soft attention with Stop and start state
Fully Differentiable	Not Fully Differentiable	Fully Differentiable	Fully Differentiable
Fixed steps	Not-Fixed steps	Not-Fixed steps	Not-Fixed steps
What only	What and Where	What only	What and Where

Dataset	**Ours**	**Baseline-CNN**	**GoodFellow** *et al.* [13]	**Ba** *et al.* [14]
SVHN	91.0%	94.3%	96.0%	96.1%
CAPTCHA	86.0%	99.8%	99.9%	-

Table 5.2: Mean sequence accuracy results on multi-digit classification and CAPTCHA tasks.

Dataset	**IOU-Score**
SVHN	81.0 %
CAPTCHA	90.0 %

Table 5.3: The sequence IOU score for the predicted bounding boxes with respect to the ground-truth bounding boxes.

Figure 5.6: Qualitative results for our method on SVHN Dataset

Figure 5.7: Qualitative results for our method on CAPTCHA Dataset

Figure 5.8: Soft attention mechanism on SVHN test dataset. Our model attends to different spatial locations in the image. The white regions correspond to the attended regions. The small white rectangle in the bottom right corner on the last image indicates a stop state.

Figure 5.9: Soft attention mechanism on CAPTCHA test dataset. Our model attends to different spatial locations in the image. The white regions correspond to the attended regions. The small white rectangle in the bottom right corner on the last image indicates a stop state.

CHAPTER 6

Conclusion

The thesis studies an attention approach to character (digits+letters) sequence prediction. The model uses attention to focus on relevant locations in the image and then uses the features extracted from these regions for digit classification. The image is passed through a CNN. Next, at each recurrent step the attention module attends to one character, starting with the left most character. The process continues until no more characters are left to read and the attention module reaches the stop state. In this work, we use attention loss to train the network to attend to the relevant regions in the image in a left-to-right fashion.

We have evaluated this approach on two digit (letter) sequence classification problems. First, we study the performance of this approach on the SVHN dataset. While our results are inferior to other techniques, the proposed method has the potential to do better. Next, we study the performance of our approach on CAPTCHA dataset. We end up creating our own CAPTCHA dataset for this work, since we did not have access to the CAPTCHA dataset used in previous approaches. Our results on CAPTCHA dataset are also not as good as those achieved by an existing tech-

nique. We suspect that this is due to the fact that classification module used for CAPTCHA letter classification is sensitive to rotations and deformations. In the future, we plan to address this issue. One idea is to use STN network to improve letter classification accuracy.

This work presented in this thesis supports our hypothesis that attention itself can play a powerful role in creating neural network models that can attend to relevant regions in the image to carry out useful work. In the future, we plan to apply this technique on other computer vision problems, such as pose estimation and action recognition.

Bibliography

[1] S. J. Russell and P. Norvig, *Artificial Intelligence - A Modern Approach, Third International Edition*. Pearson Education, 2010.

[2] Y. Zhong, R. Arandjelovic, and A. Zisserman, "Ghostvlad for set-based face recognition," in *Proc. Asian Conference on Computer Vision (ACCV)*, Perth, December 2018, pp. 35–50.

[3] Q. Zhao, T. Sheng, Y. Wang, Z. Tang, Y. Chen, L. Cai, and H. Ling, "M2det: A single-shot object detector based on multi-level feature pyramid network," in *Proc. Conference on Artificial Intelligence (AAAI)*, Honolulu, January 2019, pp. 9259–9266.

[4] Y. Xue and N. Ray, "Cell detection in microscopy images with deep convolutional neural network and compressed sensing," *arXiv preprint arXiv:1708.03307*, 2017.

[5] J. Redmon, S. K. Divvala, R. B. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," in *Proc. Conference on Computer Vision and Pattern Recognition (CVPR)*, Las Vegas, June 2016, pp. 779–788.

[6] S. Ren, K. He, R. B. Girshick, and J. Sun, "Faster R-CNN: towards real-time object detection with region proposal networks," in *Proc. Advances in Neural Information Processing Systems (NeurIPS)*, Montreal, December 2015, pp. 91–99.

[7] K. He, G. Gkioxari, P. Dollár, and R. Girshick, "Mask R-CNN," in *Proc. IEEE Conference on International Conference on Computer Vision (ICCV)*, Venice, October 2017, pp. 2980–2988.

[8] C. Vondrick, A. Shrivastava, A. Fathi, S. Guadarrama, and K. Murphy, "Tracking emerges by colorizing videos," in *Proc. European Conference on Computer Vision (ECCV)*, Munich, September 2018, pp. 391–408.

[9] R. R. Hoy, "Startle, categorical response, and attention in acoustic behavior of insects," *Annual review of neuroscience*, vol. 12, no. 1, pp. 355–375, 1989.

[10] T. R. Zentall, "Selective and divided attention in animals," *Behavioural Processes*, vol. 69, no. 1, pp. 1–15, 2005.

[11] F. Wang and D. M. Tax, "Survey on the attention based rnn model and its applications in computer vision," *arXiv preprint*, 2016.

[12] K. Xu, J. Ba, R. Kiros, K. Cho, A. C. Courville, R. Salakhutdinov, R. S. Zemel, and Y. Bengio, "Show, attend and tell: Neural image caption generation with visual attention," in *Proc. International Conference on Machine Learning, (ICML)*, Lille, July 2015, pp. 2048–2057.

[13] I. J. Goodfellow, Y. Bulatov, J. Ibarz, S. Arnoud, and V. Shet, "Multi-digit number recognition from street view imagery using deep convolutional neural networks," *arXiv preprint arXiv:1312.6082*, 2013.

[14] J. Ba, V. Mnih, and K. Kavukcuoglu, "Multiple Object Recognition with Visual Attention," in *Proc. International Conference on Robotics and Automation (ICRA)*, San Diego, May 2015.

[15] R. J. Williams, "Simple statistical gradient-following algorithms for connectionist reinforcement learning," *Machine Learning*, vol. 8, pp. 229–256, November 1992.

[16] T. Joseph, K. G. Derpanis, and F. Z. Qureshi, "Joint spatial and layer attention for convolutional networks," in *Proc. British Machine Vision Association (BMVC)*, Cardiff, September 2019, pp. 1–14.

[17] F. Wang, M. Jiang, C. Qian, S. Yang, C. Li, H. Zhang, X. Wang, and X. Tang, "Residual Attention Network for Image Classification," in *Proc. IEEE Conference on Conference on Computer Vision and Pattern Recognition (CVPR)*, Honolulu, July 2017, pp. 6450–6458.

[18] R. Ghaeini, X. Z. Fern, H. Shahbazi, and P. Tadepalli, "Saliency learning: Teaching the model where to pay attention," in *Proc. 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL-HLT)*, Minneapolis, June 2019, pp. 4016–4025.

[19] I. Goodfellow, Y. Bengio, and A. Courville, *Deep learning*. MIT press, 2016.

[20] R. S. Sutton and A. G. Barto, *Introduction to reinforcement learning*. MIT press Cambridge, 1998.

[21] F.-F. Li, J. Justin, and S. Yeung. (2018) Course notes of cs231n: Convolutional neural networks for visual recognition. [Online]. Available: http://cs231n.stanford.edu/

[22] T. Joseph, "Joint spatial and layer attention for convolutional networks," Master's thesis, University of Ontario Institute of Technology, 2018.

[23] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," in *Proc. Conference on Learning Representations (ICLR)*, San Diego, 2015.

[24] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," in *Proc. Conference on on Machine Learning (ICML)*, Lille, 2015, pp. 448–456.

[25] S. Hochreiter and J. Schmidhuber, "Long Short-Term Memory," *Neural Computation*, pp. 1735–1780, November 1997.

[26] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, http://www.deeplearningbook.org.

[27] S. Amari, "Backpropagation and stochastic gradient descent method," *Neurocomputing*, vol. 5, no. 3, pp. 185–196, 1993.

[28] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," in *Proc. International Conference on Learning Representations (ICLR)*, San Diego, May 2015.

[29] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," in *Proc. Artificial Intelligence and Statistics (AISTATS)*, Sardinia, May 2010, pp. 249–256.

[30] J. E. Moody, S. J. Hanson, and R. Lippmann, "Advances in Neural Information Processing Systems 4," in *Proc. Advances in Neural Information Processing Systems (NeurIPS)*, Denver, December 1992.

[31] N. Srivastava, G. E. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: a simple way to prevent neural networks from overfitting," *J. Mach. Learn. Res.*, pp. 1929–1958, Mach 2014.

[32] D. G. Lowe, "Distinctive image features from scale-invariant keypoints," *International Journal of Computer Vision*, vol. 60, no. 2, pp. 91–110, November 2004.

[33] H. Bay, A. Ess, T. Tuytelaars, and L. Van Gool, "Speeded-up robust features (SURF)," *Computer Vision and Image Understanding*, vol. 110, no. 3, pp. 346–359, July 2008.

[34] M. Calonder, V. Lepetit, C. Strecha, and P. Fua, "BRIEF: binary robust independent elementary features," in *Proc. 11th European Conference on Computer Vision (ECCV)*, Heraklion, September 2010, pp. 778–792.

[35] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Proc. Advances in Neural Information Processing Systems (NeurIPS)*, Lake Tahoe, December 2012, pp. 1097–1105.

[36] J. Long, E. Shelhamer, and T. Darrell, "Fully convolutional networks for semantic segmentation," in *Proc. IEEE conference on computer vision and pattern recognition (CVPR)*, Boston, June 2015, pp. 3431–3440.

[37] M. Jaderberg, K. Simonyan, A. Vedaldi, and A. Zisserman, "Deep structured output learning for unconstrained text recognition," *arXiv preprint arXiv:1412.5903*, 2014.

[38] O. Vinyals, A. Toshev, S. Bengio, and D. Erhan, "Show and tell: A neural image caption generator," in *Proc. IEEE conference on computer vision and pattern recognition (CVPR)*, Boston, June 2015, pp. 3156–3164.

[39] R. B. Girshick, J. Donahue, T. Darrell, and J. Malik, "Rich feature hierarchies for accurate object detection and semantic segmentation," in *Proc. Computer Vision and Pattern Recognition (CVPR)*, Columbus, June 2014, pp. 580–587.

[40] R. B. Girshick, "Fast R-CNN," in *Proc. IEEE international conference on computer vision (ICCV)*, Santiago, December 2015, pp. 1440–1448.

[41] D. P. Kingma and M. Welling, "Auto-encoding variational bayes," in *Proc. International Conference on Learning Representations (ICLR)*, Banff, April 2014.

[42] E. Jang, S. Gu, and B. Poole, "Categorical reparameterization with gumbel-softmax," in *Proc. of the International Conference on Learning Representations (CLR), Toulon*, April 2017.

[43] J. Kim, S. Lee, D. Kwak, M. Heo, J. Kim, J. Ha, and B. Zhang, "Multimodal residual learning for visual QA," in *Proc. Advances in Neural Information Processing Systems (NeurIPS)*, Barcelona, December 2016, pp. 361–369.

[44] R. K. Srivastava, K. Greff, and J. Schmidhuber, "Training very deep networks," in *Proc. Advances in Neural Information Processing Systems (NeurIPS)*, Montreal, December 2015, pp. 2377–2385.

[45] H. Larochelle and G. E. Hinton, "Learning to combine foveal glimpses with a third-order boltzmann machine," in *Proc. Advances in Neural Information Processing Systems (NeurIPS)*, Vancouver, December 2010, pp. 1243–1251.

[46] L. Meng, B. Zhao, B. Chang, G. Huang, F. Tung, and L. Sigal, "Where and when to look? spatio-temporal attention for action recognition in videos," *CoRR*, 2018.

[47] Y. Netzer, T. Wang, A. Coates, A. Bissacco, B. Wu, and A. Ng, "Reading digits in natural images with unsupervised feature learning," in *Proc. Advances in Neural Information Processing Systems (NeurIPS)*, 01 2011.

[48] V. Mnih, N. Heess, A. Graves, and K. Kavukcuoglu, "Recurrent Models of Visual Attention," in *Proc. Advances in Neural Information Processing Systems (NeurIPS)*, Montreal, December 2014, pp. 2204–2212.

[49] M. Jaderberg, K. Simonyan, A. Zisserman, and K. Kavukcuoglu, "Spatial transformer networks," in *Proc. Advances in Neural Information Processing Systems (NeurIPS)*, Montreal, December 2015, pp. 2017–2025.

[50] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: A System For Large-Scale Machine Learning," in *Proc. Operating Systems: Design and Implementation (OSDI)*, vol. 16, November 2016, pp. 265–283.

[51] Y. LeCun, L. Bottou, Y. Bengio, P. Haffner *et al.*, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

Appendices

APPENDIX A

Qualitative Results

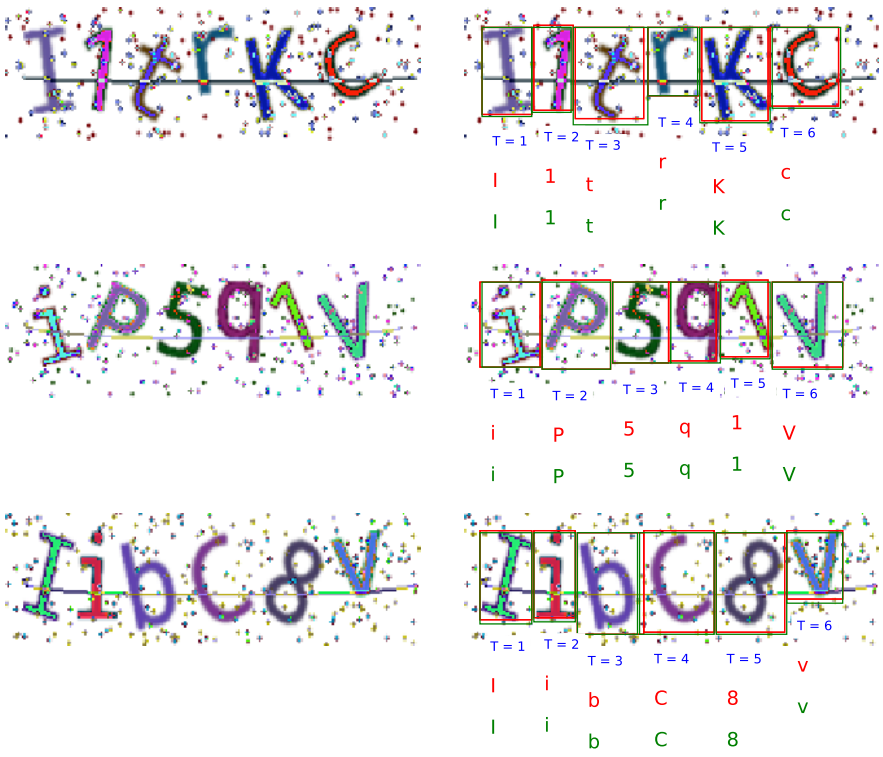


Figure A.1: Qualitative results on CAPTCHA Dataset. Ground truth bounding box is shown in green and the prediction bounding box is shown in red.

Figure A.2: Qualitative results on CAPTCHA Dataset. Ground truth bounding box is shown in green and the prediction bounding box is shown in red.

Figure A.3: Qualitative results on SVHN Dataset. Ground truth bounding box is shown in green and the prediction bounding box is shown in red.

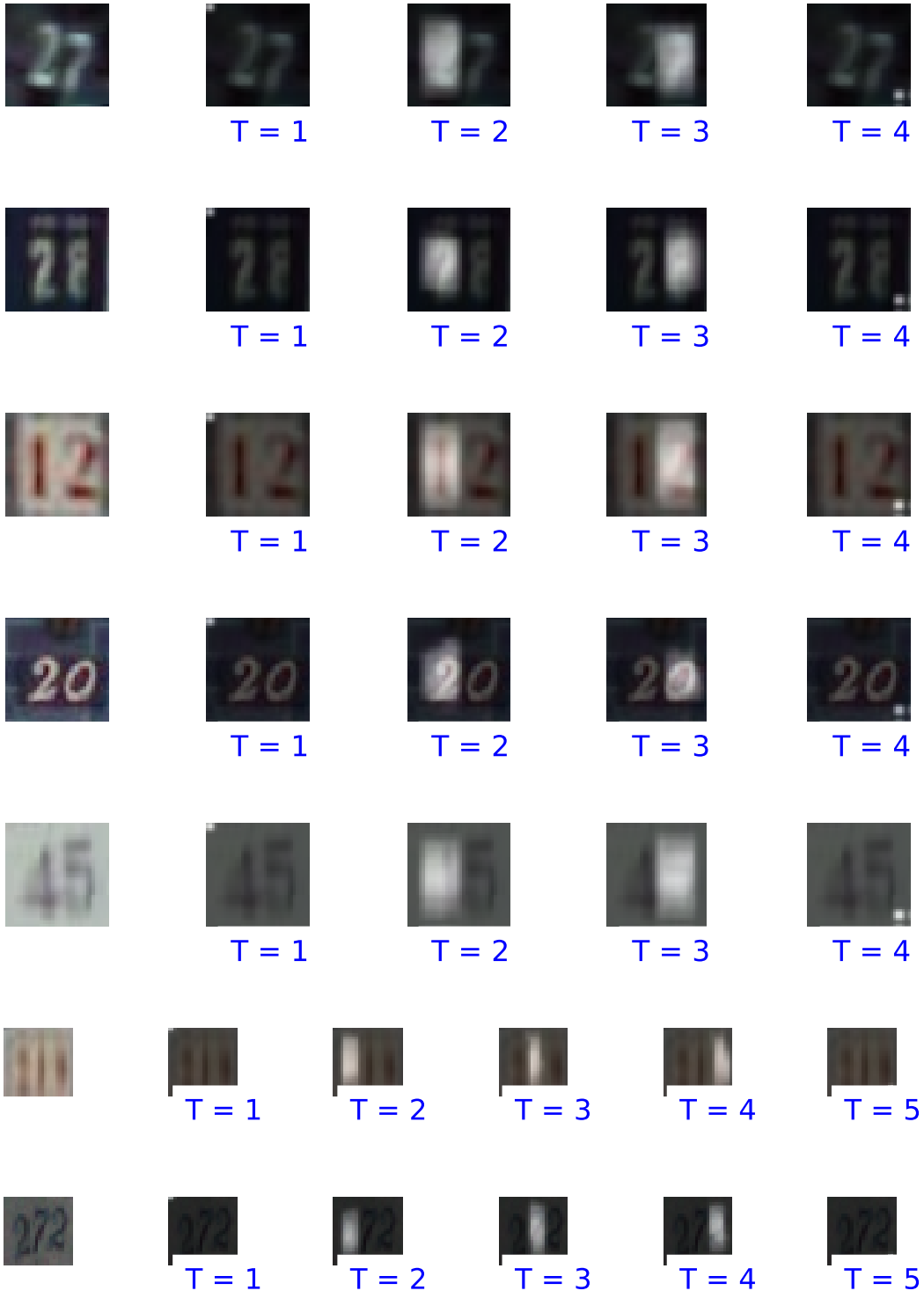
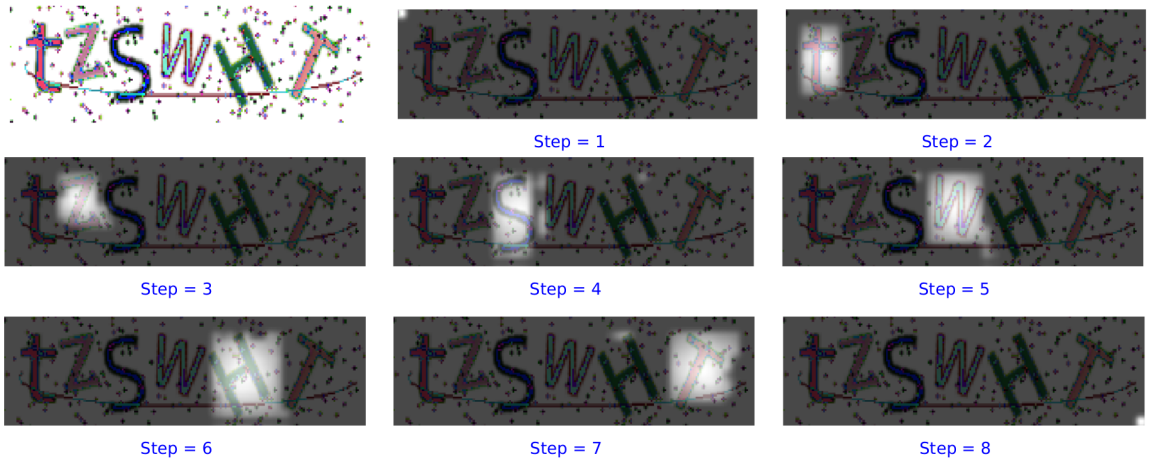
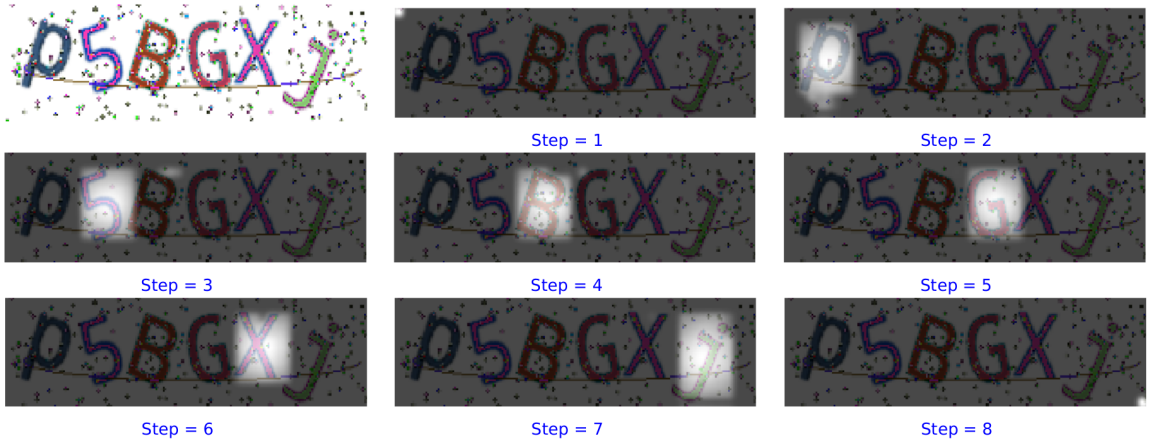


Figure A.4: Soft attention mechanism on SVHN test dataset. White regions indicate attended regions.



(a)

(b)



(c)

Figure A.5: Soft attention mechanism on CAPTCHA test dataset. White regions indicate attended regions.

APPENDIX B

Code Listings

B.1 Feature Extractor (TensorFlow Implementation)

```
#!/usr/bin/env python
# Import necessary packages
import tensorflow as tf

def cnn(images, mode=True):
    with tf.variable_scope('CNN'):
        layer_1 = slim.conv2d(images, 64, [5, 5],
                              activation_fn=None,
                              padding='VALID',
                              weights_initializer=tf.contrib.layers.
                                  variance_scaling_initializer(mode='
                                  FAN_IN'),
                              stride=1, scope='layer_1')
        layer_1 = _batch_norm(layer_1, mode=mode, name='layer_1')
        layer_1 = tf.nn.leaky_relu(layer_1,  name='relu_layer_1')

        layer_2 = slim.conv2d(layer_1, 96, [5, 5],
```

```
16                              activation_fn=None,
17                              padding='VALID',
18                              weights_initializer=tf.contrib.layers.
                                    variance_scaling_initializer(mode='
                                    FAN_IN'),
19                              stride=1, scope='layer_2')
20          layer_2 = _batch_norm(layer_2, mode=mode, name='layer_2')
21          layer_2 = tf.nn.leaky_relu(layer_2,  name='relu_layer_2')
22
23          layer_3 = slim.conv2d(layer_2, 128, [5, 5],
24                              activation_fn=None,
25                              padding='SAME',
26                              weights_initializer=tf.contrib.layers.
                                    variance_scaling_initializer(mode='
                                    FAN_IN'),
27                              stride=2, scope='layer_3')
28          layer_3 = _batch_norm(layer_3, mode=mode, name='layer_3')
29          layer_3 = tf.nn.leaky_relu(layer_3,  name='relu_layer_3')
30
31          layer_4 = slim.conv2d(layer_3, 128, [5, 5],
32                              activation_fn=None,
33                              padding='SAME',
34                              weights_initializer=tf.contrib.layers.
                                    variance_scaling_initializer(mode='
                                    FAN_IN'),
35                              stride=1, scope='layer_4')
36          layer_4 = _batch_norm(layer_4, mode=mode, name='layer_4')
37          layer_4 = tf.nn.leaky_relu(layer_4,  name='relu_layer_4')
38
39          layer_5 = slim.conv2d(layer_4, 128, [5, 5],
40                              activation_fn=None,
41                              padding='SAME',
42                              weights_initializer=tf.contrib.layers.
                                    variance_scaling_initializer(mode='
                                    FAN_IN'),
43                              stride=1, scope='layer_5')
```

```
44          layer_5 = _batch_norm(layer_5, mode=mode, name='layer_5')
45          layer_5 = tf.nn.leaky_relu(layer_5,  name='relu_layer_5')
46
47          layer_6 = slim.conv2d(layer_5, 128, [5, 5],
48                                activation_fn=None,
49                                padding='SAME',
50                                weights_initializer=tf.contrib.layers.
                                      variance_scaling_initializer(mode='
                                      FAN_IN'),
51                                stride=2, scope='layer_6')
52          layer_6 = _batch_norm(layer_6, mode=mode, name='layer_6')
53          layer_6 = tf.nn.leaky_relu(layer_6,  name='relu_layer_6')
54
55          layer_7 = slim.conv2d(layer_6, 128, [5, 5],
56                                activation_fn=None,
57                                padding='SAME',
58                                weights_initializer=tf.contrib.layers.
                                      variance_scaling_initializer(mode='
                                      FAN_IN'),
59                                stride=1, scope='layer_7')
60          layer_7 = _batch_norm(layer_7, mode=mode, name='layer_7')
61          layer_7 = tf.nn.leaky_relu(layer_7,  name='relu_layer_7')
62
63          layer_8 = slim.conv2d(layer_7, 128, [3, 3],
64                                activation_fn=None,
65                                padding='SAME',
66                                weights_initializer=tf.contrib.layers.
                                      variance_scaling_initializer(mode='
                                      FAN_IN'),
67                                stride=1, scope='layer_8')
68          layer_8 = _batch_norm(layer_8, mode=mode, name='layer_8')
69
70          return layer_8, layer_4
```

Listing B.1: TensorFlow implementation of the feature extractor (convolutional network) in python

B.2 Spatial Transformer Network (TensorFlow Implementation)

```
#!/usr/bin/env python
with tf.variable_scope('Localisation network'):
  B1, H1, W1, C1 = inputs.get_shape().as_list()
  fln_inputs = tf.reshape(inputs, [-1, H1*W1*C1])
  _, D = fln_inputs.get_shape().as_list()
  with tf.variable_scope(name_scope, reuse=reuse):
    w = tf.get_variable(shape=[D, 6], initializer=self.const_initializer
        , name='weights')
    b = tf.get_variable(shape=[6],    initializer=self.ident_initializer
        , name='biases')
    theta  = tf.nn.tanh(tf.matmul(fln_inputs, w) + b)
```

Listing B.2: Tensorflow implementation of the localisation network of the STN in python

B.3 Attention Module (TensorFlow Implementation)

```
#!/usr/bin/env python
def attention_layer(self, features, h, reuse=False):
    with tf.variable_scope('attention_layer', reuse=reuse):
        w = tf.get_variable(shape=[self.H, self.D], initializer=self.
            weight_initializer, name='weights')
        b = tf.get_variable(shape=[self.D], initializer=self.
            const_initializer, name='biases')
        w_att = tf.get_variable(shape=[self.D, 1], initializer=self.
            weight_initializer, name='w_weights')

        h_att   = tf.nn.relu(features + tf.expand_dims(tf.matmul(h, w),
            1) + b) # (N, L, D)
```

```
        out_att = tf.reshape(tf.matmul(tf.reshape(h_att, [-1, self.D]),
            w_att), [-1, self.L]) # (N, L)
        alpha   = tf.nn.softmax(out_att)
        out_ft  = features * tf.expand_dims(alpha, 2)
        context = tf.reduce_sum(out_ft, 1, name='context') #(N, D)

 return context
```

Listing B.3: Tensorflow implementation of the attention module in python

B.4 Grid Generation for STN (TensorFlow Implementation)

```
#!/usr/bin/env python
def meshgrid(height, width):
    with tf.variable_scope('grid'):
        x_t = tf.matmul(tf.ones(shape=tf.stack([height, 1])),
                        tf.transpose(tf.expand_dims(tf.linspace(-1.0,
                            1.0, width), 1), [1, 0]))
        y_t = tf.matmul(tf.expand_dims(tf.linspace(-1.0, 1.0, height),
            1),
                        tf.ones(shape=tf.stack([1, width])))

        x_t_flat = tf.reshape(x_t, (1, -1))
        y_t_flat = tf.reshape(y_t, (1, -1))

        ones = tf.ones_like(x_t_flat)
        grid = tf.concat(axis=0, values=[x_t_flat, y_t_flat, ones])
        return grid

def transform(theta, input_dim, out_size):
    with tf.variable_scope('_transform'):
        num_batch = tf.shape(input_dim)[0]
```

```
        height = tf.shape(input_dim)[1]
        width = tf.shape(input_dim)[2]
        num_channels = tf.shape(input_dim)[3]
        theta = tf.reshape(theta, (-1, 2, 3))
        theta = tf.cast(theta, 'float32')
        # grid of (x_t, y_t, 1),
        height_f = tf.cast(height, 'float32')
        width_f = tf.cast(width, 'float32')
        out_height = out_size[0]
        out_width = out_size[1]
        grid = _meshgrid(out_height, out_width)
        grid = tf.expand_dims(grid, 0)
        grid = tf.reshape(grid, [-1])
        grid = tf.tile(grid, tf.stack([num_batch]))
        grid = tf.reshape(grid, tf.stack([num_batch, 3, -1]))

        # Transform grid using the predicted theta:
        #             Theta x (x_t, y_t, 1)^T -> (x_s, y_s)
        T_g = tf.matmul(theta, grid)
        x_s = tf.slice(T_g, [0, 0, 0], [-1, 1, -1])
        y_s = tf.slice(T_g, [0, 1, 0], [-1, 1, -1])
        x_s_flat = tf.reshape(x_s, [-1])
        y_s_flat = tf.reshape(y_s, [-1])

    return x_s_flat, y_s_flat
```

Listing B.4: Tensorflow implementation of the grid generator of the STN in python

B.5 Sampler for STN (TensorFlow Implementation)

```
#!/usr/bin/env python
def sampler(im, x, y, out_size):
    with tf.variable_scope('sampler'):
        # constants
```

```
        num_batch = tf.shape(im)[0]
        height = tf.shape(im)[1]
        width = tf.shape(im)[2]
        channels = tf.shape(im)[3]

        x = tf.cast(x, 'float32')
        y = tf.cast(y, 'float32')
        height_f = tf.cast(height, 'float32')
        width_f = tf.cast(width, 'float32')
        out_height = out_size[0]
        out_width = out_size[1]
        zero = tf.zeros([], dtype='int32')
        max_y = tf.cast(tf.shape(im)[1] - 1, 'int32')
        max_x = tf.cast(tf.shape(im)[2] - 1, 'int32')

        # scale indices from [-1, 1] to [0, width/height]
        x = (x + 1.0)*(width_f) / 2.0
        y = (y + 1.0)*(height_f) / 2.0

        # do sampling
        x0 = tf.cast(tf.floor(x), 'int32')
        x1 = x0 + 1
        y0 = tf.cast(tf.floor(y), 'int32')
        y1 = y0 + 1

        x0 = tf.clip_by_value(x0, zero, max_x)
        x1 = tf.clip_by_value(x1, zero, max_x)
        y0 = tf.clip_by_value(y0, zero, max_y)
        y1 = tf.clip_by_value(y1, zero, max_y)
        dim2 = width
        dim1 = width*height
        base = _repeat(tf.range(num_batch)*dim1, out_height*out_width)
        base_y0 = base + y0*dim2
        base_y1 = base + y1*dim2
        idx_a = base_y0 + x0
        idx_b = base_y1 + x0
```

```
        idx_c = base_y0 + x1
        idx_d = base_y1 + x1

        # use indices to lookup pixels in the flat image and restore
        # channels dim
        im_flat = tf.reshape(im, tf.stack([-1, channels]))
        im_flat = tf.cast(im_flat, 'float32')
        Ia = tf.gather(im_flat, idx_a)
        Ib = tf.gather(im_flat, idx_b)
        Ic = tf.gather(im_flat, idx_c)
        Id = tf.gather(im_flat, idx_d)

        # and finally calculate interpolated values
        x0_f = tf.cast(x0, 'float32')
        x1_f = tf.cast(x1, 'float32')
        y0_f = tf.cast(y0, 'float32')
        y1_f = tf.cast(y1, 'float32')
        wa = tf.expand_dims(((x1_f-x) * (y1_f-y)), 1)
        wb = tf.expand_dims(((x1_f-x) * (y-y0_f)), 1)
        wc = tf.expand_dims(((x-x0_f) * (y1_f-y)), 1)
        wd = tf.expand_dims(((x-x0_f) * (y-y0_f)), 1)
        output = tf.add_n([wa*Ia, wb*Ib, wc*Ic, wd*Id])

    return output
```

Listing B.5: Tensorflow implementation of the sampler of the STN in python

B.6 CAPTCHA Generation

```
import cv2
import string
import random
import numpy as np
from PIL import Image
```

```
from PIL.ImageDraw import Draw
from captcha import ImageCaptcha
from captcha import random_color
# Seed
#np.random.seed(8964)
#----------Functions----------
# Random dots generator
def create_noise_dots(image, color, width=2, number=125):
    draw = Draw(image)
    w, h = image.size
    while number:
        x1 = random.randint(0, w)
        y1 = random.randint(0, h)
        draw.line(((x1, y1), (x1 - 1, y1 - 1)), fill=color, width=width)
        number -= 1
    return image
# Random curve generator
def create_noise_curve(image, color):
    w, h = image.size
    x1 = random.randint(10, 15)
    x2 = random.randint(w - 10, w)
    y1 = random.randint(20, 35)
    y2 = random.randint(y1, 60)
    points = [x1, y1, x2, y2]
    end = random.randint(180, 200)
    start = random.randint(0, 20)
    Draw(image).arc(points, start, end, fill=color)
    return image
# Caption generator
image = ImageCaptcha(width=64, height=64, font_sizes=(64, 64, 64))
#------------------------------
# Get characters
az = string.ascii_lowercase
AZ = string.ascii_uppercase
nm = string.digits
#------------------------------
```

```
42 # Append all characters
43 all_selections = []
44 for i in range(len(az)):
45     all_selections.append(az[i])
46 for i in range(len(AZ)):
47     all_selections.append(AZ[i])
48 for i in range(len(nm)):
49     all_selections.append(nm[i])
50 #----------------------------
51
52 #-------------MAIN-------------
53 count = 0
54 max_chars = 6
55 train_set = []
56 dump_file = './dataset/captcha/captcha.txt'
57
58 # Atleast have all characters appear once in trainset -- generate 100 for
       each
59 for select in all_selections:
60     counter1 = 0
61     while counter1 < 500:
62         all_data = []
63         all_lbls = []
64         # Randomly generate a placeholder to put in the char
65         insert_id  = random.randint(0, max_chars)
66         # Generate numbers
67         max_steps = random.randint(4, max_chars)
68         for t in range(max_steps):
69             if t == insert_id:
70                 lbl  = select
71                 data = image.generate_image(select)
72             else:
73                 idx  = random.randint(0, len(all_selections)-1)
74                 lbl  = all_selections[idx]
75                 data = image.generate_image(lbl)
76             # Append
```

```
77              all_lbls.append(lbl)
78              all_data.append(data)
79          # Get max width
80          total_w = 0
81          for i in all_data:
82              pix = np.array(i)
83              h, w, _ = pix.shape
84              total_w += w
85          # Get max height
86          highest_h = 50
87          for i in all_data:
88              pix = np.array(i)
89              h, w, _ = pix.shape
90              if h < highest_h:
91                  highest_h = h
92          # Begin painting
93          canvas = np.ones((highest_h + 35, total_w + 35, 3)).astype(np.
                uint8) * 255
94          prev_w = 10
95          all_bbox = []
96          try:
97              for i in all_data:
98                  pix = np.array(i)
99                  h, w, _ = pix.shape
100                 # print(pix.shape)
101                 # Get BBox's
102                 h1 = random.randint(8, 20)
103                 w1 = prev_w
104                 h2 = h
105                 w2 = w
106                 all_bbox.append([w1, h1, w2, h2])
107                 # Paint Canvas
108                 canvas[h1:h1+h, prev_w+2:prev_w+2+w, :] = pix
109                 prev_w += w
110             # Append to training set
111             train_set.append([all_lbls, all_bbox])
```

```
            # Convert to PIL Image
            im = Image.fromarray(canvas)
            # Generate different colored dots
            color = random_color(10, 200, random.randint(220, 255))
            im = create_noise_dots(im, color)
            color = random_color(10, 200, random.randint(220, 255))
            im = create_noise_dots(im, color)
            color = random_color(10, 200, random.randint(220, 255))
            im = create_noise_curve(im, color)
            # Convert to numpy array
            canvas = np.array(im)
            # Save image
            cv2.imwrite('./dataset/captcha/dataset/%d.png' % count,
                canvas)
            # Increment
            count += 1
            # Progress
            if count%500 == 0:
                print('Creating dataset --Progress: ', count)
            counter1 += 1
        except ValueError:
            print('Skipping.....')
            continue
```

Listing B.6: Python implementation of the CAPTCHA generation code