Neural Network CAPTCHA Crackers

Geetika Garg, Chris Pollett

Abstract—This paper describes several experiments using deep neural networks to break character-based image CAPTCHAs. The goal of our research was to see if one could develop a single neural network capable of breaking all character-based, image CAPTCHAs. Our main deep neural net uses convolutional neural network layers followed by a dense layer, and a recurrent recurrent neural network layer instead of the conventional method of CAPTCHA breaking based on segmenting and recognizing individual letters. Our experiments with these networks were conducted using a synthetically generated dataset of CAPTCHAs which is independently useful for future research. We trained on both fixed-and variable-length CAPTCHAs and our main neural net configuration was able to achieve accuracy levels of 99.8% and 81%, respectively.

Keywords—Image-based CAPTCHAs, deep neural networks

I. Introduction

A Completely Automated Public Turing test to tell Computers and Humans Apart, or CAPTCHA, is a problem that is very easy for a human to solve, but very difficult for a computer to solve. [1] Typically, it involves a task like recognizing a string of characters in an image. In this paper, we describe our work to make it easier for a computer to solve string-of-character CAPTCHAs.

When CAPTCHAs were first proposed, there were no AI programs that could easily perform the tasks necessary to break them using computer vision techniques. Recently, deep neural networks have brought major advancements in the field of AI and computer vision. They have reached state-of-the-art or better performance as compared to other methods in the fields of speech recognition [9], computer vision, natural language processing [17], and language translation [26]. It is therefore interesting to see to what extent deep neural networks can be used to break CAPTCHAs to assess how secure CAPTCHA-based security systems currently are.

Traditionally, for object and character recognition tasks in computer vision, separate modules were created for preprocessing (noise reduction), character segmentation, character recognition, and sequence generation, where the sequence of characters with the highest probability was generated. Splitting by character, however, means that the system cannot easily use information from the whole image to help in the recognition task. The error rate on the whole task is at least the error rate for a single character which is turn at least the error rate of any individual step in the process of recognizing a character. On the other hand, the system we describe in this paper is trained on the whole image and can make use of the information between characters.

Geetika Garg is a Software Engineer with Cisco Systems, 725 Alder Drive, Milpitas CA (email: geetikagarg07@gmail.com)

Chris Pollett is a professor in the Department of Computer Science, San Jose State University, 1 Washington Square, San Jose CA 95192 (e-mail: chris@pollett.org).

The use of whole images is motivated by how humans might solve CAPTCHAs. We know the human mind is capable of recognizing strings of characters, and the process by which this is done has been elucidated over the years by many remarkable psychological experiments involving timing tests, brain imaging, etc. [4]. The clearest visual inputs come from the fovea of the eye as it saccades over a piece text. McConkie and Rayner [16] have shown that how the eye moves in its saccades is adapted to the size of the text, and that it is typically on the order of seven to nine characters – not single characters. This is also about the length of longest CAPTCHAs, and suggestive that a CAPTCHA would be treated as a whole unit by the mind. Typically, when words are involved the gaze is focussed slightly to the left of the current word being read, and various differences between the characters in the word, such as font variants in a letter like 'r' are ignored, while other important difference, such as between e and o are kept. This suggests that a program to break CAPTCHAs should have units, such as convolutional layers, capable of computing these invariants and differences.

The system we present is an example of a so-called deep neural network. That is, it uses an artificial neural network of consisting of multiple layers where the layers have particular connection topologies which are less than the full set of connections between the given layer and the prior layer or inputs. The fact that the layers are less than fully connected means that the space of weights that need to be trained for using back propagation and related algorithms is smaller, so the rate of convergence of our network will typically be faster than for a fully connected network, and since the space of weights is smaller, the risk of overfitting is reduced.

Our work, developed initially as part of the first author's master's writing project, combines the idea of two papers, [8] and [28], which guided our choice of layers to use in building our deep neural network. We built a model that uses convolutional neural network layers to learn CAPTCHA image features and then uses those features in a recurrent neural network layer followed by a softmax layer to output the sequence of characters of an image. The combination of these layers gives a single deep neural network that performs end-to-end CAPTCHA recognition. For our recurrent neural network layer, we used an LSTM (Long Short Term Memory) layer [11] [12] [13].

The rest of this paper is organized as follows: In the next section, we briefly discuss the history of CAPTCHAs and their use in online spam prevention. This is followed by a section detailing prior work on image-based, CAPTCHA breaking. After this, we have a section presenting the necessary background for our results with respect to general neural networks, network training, and neural network layer types. We then describe the CAPTCHA dataset we created for our

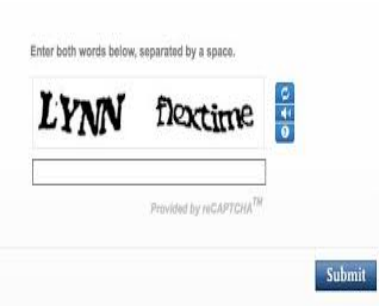

Fig. 1 A Sample image-based CAPTCHA

experiments. We give the neural network models we used in our CAPTCHA breaking experiments. We then describe our experiments. In the last section, we draw some conclusions from our experiments.

II. Introduction to CAPTCHAs

In this section we review the important aspects of CAPTCHAs that are needed to understand our decisions in designing neural networks to break them. CAPTCHAs were first mentioned in an unpublished paper by Moni Naor [20] in 1996, but were popularized by the paper of Von Anh, et al [1]. CAPTCHAs were first used on an actual website by AltaVista and were patented by the company that owned AltaVista, Compaq [2]. A character-based image CAPTCHA is an image presented on a web page containing a series of oddly drawn letters or numbers together with a text field in which a user is required to enter the characters presented in the image. Correctly, entering the sequence allows the user to continue carrying out some activity on the given web page. For example, a CAPTCHA might appear at the end of a form to sign up for a free email account. Entering the CAPTCHA correctly would allow the user to sign up, answering incorrectly would mean no account is set up. If the CAPTCHA is hard enough for a computer it could be used to prevent a email spam bot from signing up for an account and sending out spam emails.

Figure 1 gives an example of a character-based image CAPTCHA. CAPTCHAs are based on the assumption that the task required is something that is difficult for a computer to do. That is, it is a difficult AI problem. Thus, teaching computers to break CAPTCHAs is work towards solving the associated AI problem. This results in a win-win situation: So as long as a CAPTCHA is hard, it can be used to prevent spam, and when it is broken, the resulting algorithm can be used for other problems in computer vision. In our particular case, the same kind of neural nets as we develop to break CAPTCHAs could be used for word recognition. Breaking more recent kinds of CAPTCHAs, which involve having the subject click on all images belonging to some category such as a particular food type, would further help with image categorization algorithms.

At this point we should clarify what it means to break a CAPTCHA. In most CAPTCHA systems, a user is allowed some number of tries to solve the CAPTCHA. After each try, the time before the next try might be lengthened, or after some number of tries, the system will refuse any further tries. A lengthening time out system can easily confuse a bot, so it is probably easier in this case to say the CAPTCHA is broken if the system can guess the CAPTCHA on the first try. In a fixed number of try system, a much lower success rate is needed to effectively break the CAPTCHA system. For example, suppose one is allowed only three tries, and one's CAPTCHA breaker has a success rate of 50%. Then the odds that the CAPTCHA is broken after three tries is $1-(1-.5)^3 \times 100\% = 87.5\%$, which is quite high. So a system which has a success rate above a half is largely compromised in the multiple try scenario.

In the single try scenario, one should take the perspective of the person using a CAPTCHA-breaking bot in deciding if the CAPTCHA is broken. If many sites use the same CAPTCHA system, as may be the case for a popular bulletin board, then one still might be able to create a fair amount of spam volume on these sites even with a mediocre CAPTCHA breaking success rate. For example, suppose one has a spam bot that discovers 100,000 sites each running the same kind of bulletin board. It then tries to post a spam message on each of these sites with a success rate of 50%. This would still result in 50,000 successful spam messages. The creators of the bulletin board system should thus view the CAPTCHA as broken because they would likely receive a lot of complaints from the sites that were compromised, and they would fear the impact of these complaints on adoption of their software.

Most bulletin boards are relatively low traffic and have a more limited impact in terms of likely web-page views than a top 100 or even top 1000 site. If a CAPTCHA-breaking bot is targeting a single site, the desire might still be to have as many successful CAPTCHA entries as possible to spread whatever message or to acquire as many accounts as one can on a large site. In this situation, the large site has a better odds against the CAPTCHA breaker as long as the CAPTCHA breaker is not running on too many IP addresses. In this setting, the single site can just monitor CAPTCHA fails from an IP address and ban that address if that number of failures ever gets too high, retroactively removing the spam messages or locking the affected accounts. Unless the CAPTCHA-breaker was coming from a botnet, this would tend to limit its reach on a major site except if the success rate was very high. It is in this kind of hard scenario that we envision our CAPTCHA-breaking system working. That is, there is a larger incentive to make the success rate as high as possible, rather than settling for "good enough". A large site would typically be trying to determine bot-like behavior even for an agent that is correctly entering CAPTCHAs. A spam-bot is a reasonably sophisticated piece of software with many components, only one of which might deal with CAPTCHA breaking. Any one of these parts might cause the bot's traffic to be flagged by the large site. The goal in this hard CAPTCHA-breaking setting is to get the CAPTCHA component to be successful enough that it is not the main cause of a spam-bot being flagged. If a CAPTCHA-breaker has at least as good performance as a human at recognizing the CAPTCHA, then it is unlikely that it will be the component to cause a flag.

Building a CAPTCHA breaker for any of the particular problems above does not entail that the concept of character-based CAPTCHAs has been **fully broken**.

For this, one would have to have a single, character-based, CAPTCHA-breaker which achieves near human-level performance on any character-based CAPTCHA problem, not just the particular problem of a particular kind of website. We view our work as taking a step towards the goal of fully breaking the character-based CAPTCHA problem.

III. Related Work

We now turn our attention to previously created CAPTCHA breaking systems. Since CAPTCHAs first became popular on the web, several such systems have been developed. For example, Mori and Malik [19] created a system which breaks the EZ-Gimpy CAPTCHA with a 92% success rate. It further breaks the Gimpy CAPTCHA with a 33% success rate, These systems were hand-tuned and used sophisticated object recognition algorithms. In comparison to these earlier works, we trained an end-to-end neural network system that can extract the features needed for classification with minimal hand-tuning. Artificial neural networks have shown great promise in many domains, such as natural language processing [17], speech [9], and image processing [26] [28]. We will discuss neural networks later in this paper, however, suffice it to say they have been used before as a component of CAPTCHA breaking systems. For now we want to consider how they have been used previously to distinguish this previous use from how we used neural nets. A representative example of how neural nets have been previously used is given in the paper [3]. In it, they describe the following steps to determine individual characters in a CAPTCHA:

1) Preprocessing
2) Segmentation
3) Training for individual character recognition

These steps are often very difficult to do because:

1) Segmentation is difficult, as some digits can overlap with other digits.
2) Understanding individual character deformity is difficult. For example, a digit "2" can have a larger loop or just a cusp.
3) Determining the scale of characters is difficult. It is not known how big a character will be, so it is not known how big the segmentation boxes should be.
4) Character orientation is difficult. Characters can be rotated at arbitrary angles, making recognition difficult.

Modules for each of the steps mentioned above are optimized independently, so systems combining these modules do not work well in practice.

Our approach is to instead learn a deep neural network, a single monolithic system for embedding these modules, and train the entire network together to make sure that that objectives of all the modules are aligned. To train our system, we can just provide CAPTCHA images and let it learn image features and how to use these features for recognition.

An example of using a similar approach to ours, but in a different context, is given by the paper [26]. There a convolutional neural network for detecting handwritten mailing addresses using convolutional neural networks was given. It achieved a 96% accuracy in recognizing complete street numbers. Their work suggested our use of convolutional network layers in our network. However, in [26], they fixed the length of the street number in an image. If one intends to build a system that can fully-break, character-based image CAPTCHAs, then one cannot assume one knows the length of the CAPTCHA in advance. To tackle this problem, we incorporated their set up as a layer that feeds into a recurrent neural network. Such networks have achieved good results recently, as shown by the "show and tell" paper [28] by Google, where they generate a caption (of variable length) for a given image. The challenging task in decoding a CAPTCHA is to guess all the symbols in the CAPTCHA correctly. If even one is wrong, we have to discard the result. In view of the papers [26] and [28], it seems natural to try to use a neural network combining a convolutional layer with a recurrent network to try to fully solve the character-based, image CAPTCHA problem. From the research mentioned in the introduction about the number of characters in focus in a human-eye saccade, one would expect that one would not need to be able to handle arbitrary variations in CAPCTHA length to achieve human-level performance on this problem. Instead, one would expect to need to only handle variation up to around seven to nine characters.

IV. Neural Network Background

Our CAPTCHA-breaker makes use of artificial neural networks so in this section we give a brief introduction to this topic. Artificial neural networks are inspired by the brain where the base computing units are cells called neurons which form various connections between each other. Computation is mainly carried out by these cells by electrical signals sent between them. In 1943, McCulloch and Pitts [15] abstracted out the key features of the biology-based neurons and considered networks built out of these artificial neurons.

A basic unit in an artificial neural network typically consists of some kind of activation function, $g : \mathbb{R} \longrightarrow \mathbb{R}$, applied to a weighted sum of input values. Nonrecurrent networks, usually known as feedforward networks, of such neurons are directed, acyclic graphs, $G = (V, E)$. In these graphs, source nodes, nodes of indegree 0, are inputs and sinks, nodes of outdegree 0, are outputs. Each internal nodes is labeled with a gate equation. Nodes are allowed to have multiple outgoing edges. Sometimes rather than using a weighted sum, gates that compute other simple functions of the inputs such as products or maxes are also considered. Given an assignment ν of a value to each input vertex, the value of the output of the network can be computed in a bottom up fashion from the inputs applying the gate equation of each node for which its inputs have already been computed. Let $\text{in}(v_j) = \{v_i | (v_i, v_j) \in E\}$. That is, if the gate equation was a weighted sum, the value of node v_i is

$$\nu(v_i) = g(\sum_{v_j \in \text{in}(v_i)} w_{ji} \cdot \nu(v_j)).$$

Here w_{ji} is a real-valued weight for the edge between node v_j and v_i. An example activation function g is a step function, which takes one value for inputs less

than or equal to the threshold and a different value for values larger than this threshold. Often it is useful to have continuous and differentiable activation functions whose behavior approximates a step function. A popular choice of such a function is the sigmoid function, a type of logistic function, $g(t) = \frac{1}{1+e^{-t}}$. Another popular choice of activation function, and the one we used in our experiments, is the rectifier function $g(t) = \max(0, t)$ which is continuous for all reals, and differentiable everywhere except the origin.

Typically, when designing artificial neural networks, neurons are arranged into layers, the 0th layer being the inputs and the last layer being the output, and where only neurons in the j layer serve as inputs to the $j + 1$st layer. If the sigmoid function is used, and the gate equations are all weighted-sums, than a single layer artificial network, a perceptron layer, computes a relatively simple, differentiable vector-valued function of its inputs. For these networks, one can use gradient-descent methods to find the "closest" such function to a set of training data. Unfortunately, it was shown by Marvin Minsky and Seymour Papert [18] that for training data coming from the output of the parity function, this "closest" perceptron layer might be correct on only half of the inputs. Because of this result, and because multi-layer networks were computationally expensive and there were no good learning algorithm for this harder setting, artificial neural networks experienced a decline in popularity in the 1960s.

In the seventies, Werbos [29] developed the backpropagation algorithm for training multi-layer neural networks based on earlier work coming from control theory. In 1986, Rumelhart, et al [23] performed experiments which showed this method produced useful internal weights in neural networks. This led to renewed interest in neural networks. As the number of layers, and so weights in these layers, grows in an artificial neural network, the amount of data needed to effectively train an artificial neural network increases. This again caused a gradual decline in popularity of neural nets as it was often difficult to obtain such large data sets.

From about 2005 onwards, several trends converged that resulted in a second resurgence of popularity of neural nets. Firstly, computers continued to get faster, both chips with many cores and graphics cards being used as general compute platforms, allowed larger problems to be effectively tackled. Secondly, computer memory also got cheaper, which allowed larger data sets to be held closer to the faster chips. Further, with the advent of the web, larger-scale, training data sets became more readily available. Finally, better models for recurrent networks, networks which relax the condition that our networks be acyclic, were developed. It has been argued that artificial neural nets are currently the hottest area in the field of machine learning [21]. Recent impressive applications of artificial neural networks include facial recognition as used by Facebook to tag photos [27] and image captioning [28] by Google.

A. Training Neural Networks

For this paper we use a variant of the Stochastic Gradient Descent (SGD) method to learn the edge-weights of neurons in an artificial neural network according to training data. We assume that the training data consists of a sequence of input-output vectors $(\vec{x}, \vec{y})$. The basic SGD algorithm starts by initializing weights for each neuron in the network to random real values within a bounded range. In the case of our networks, we set this range to $[-W_{bd}, W_{bd}]$ where W_{bd} is a function determined by the number of input and output edges of the given neuron, num_edges, such as

$$W_{bd} = \left(\frac{12}{\text{num\_edges}}\right)^{1/2}.$$

A motivation for this choice of W_{bd} can be found in Section 4.6 of Haykin [10]. Let $\vec{W}_0$ denote these randomly chosen initial weights. The algorithm proceeds in steps $k \geq 0$. During the kth step, the network's weights will be denoted, $\vec{W}_k$, one training pair $(\vec{x}_k, \vec{y}_k)$ is considered, and a new set of weights $\vec{W}_{k+1}$ is computed. Once we have cycled through all the training vectors, we say an epoch has ended, we check a stopping condition, such as the number of epochs is greater than some value, that a loss function is less than some value, or that the loss function is changing less than some value. If the condition is met the algorithms stops outputting the current weights, otherwise, it begins cycling through the training vectors again from the beginning to compute the next $\vec{W}_k$.

In the case of a single layer network, to compute $\vec{W}_{k+1}$ from $\vec{W}_k$, first let $\vec{f}(\vec{x}, \vec{W})$ denote the output of this layer as a function of its inputs and weights. A loss function $\text{Loss}(\vec{y}, \vec{x}, \vec{W}) = E(\vec{y}, \vec{f}(\vec{x}, \vec{W}))$ measures the discrepancy between the desired output $\vec{y}$ on input $\vec{x}$ and the actual $\vec{f}(\vec{x}, \vec{W})$. A simple choice for $E(\vec{y}, \vec{z})$ might be $||\vec{y} - \vec{z}||^2$, in which case, $\text{Loss}(\vec{y}, \vec{x}, \vec{W})$ is $||\vec{y} - \vec{f}(\vec{x}, \vec{W})||^2 \geq 0$. For our experiments in this paper, we chose E to be the cross entropy loss function for its better convergence properties. The cross entropy is defined as

$$E(\vec{y}, \vec{z}) = \sum_m [-y_m ln z_m].$$

Except at the origin, our activation function is differentiable and easy to compute, so the Jacobian $\frac{\partial}{\partial \vec{W}} \text{Loss}_k(\vec{y}_k, \vec{x}_k, \vec{W})$ is well-defined except for inputs which trigger the origin as value, and points in a direction to minimize the loss. So the algorithm computes

$$\vec{W}_{k+1} = \vec{W}_k - \lambda \frac{\partial \text{Loss}(\vec{y}, \vec{x}, \vec{W})}{\partial \vec{W}} \Bigg|_{\substack{\vec{y} = \vec{y}_k, \\ \vec{x} = \vec{x}_k, \\ \vec{W} = \vec{W}_k}}.$$

Here λ is the learning rate. By the chain rule,

$$\frac{\partial \text{Loss}(\vec{y}, \vec{x}, \vec{W})}{\partial \vec{W}} = \frac{\partial E}{\partial \vec{z}} \cdot \frac{\partial \vec{f}}{\partial \vec{W}}$$

where $\vec{z} = \vec{f}$. The gradient with respect to components of $\vec{z}$ is straightforward to compute for our loss function, and as $\vec{f}$ is either activation functions applied to a weighted sum, or some other easy to compute function of inputs, the partials on the right above are straightforward to compute and evaluate, thus, allowing us to compute our update rule. The variant of SGD used in this paper, SGD with Nestorov momentum, adds to

this update rule a "momentum-like" correction, which tends to yield better convergence rates [25]. Let $\vec{\Delta}_{k+1} = \vec{W}_{k+1} - \vec{W}_k$. Then our update rule is instead

$$\vec{W}_{k+1} = \vec{W}_k - \lambda \frac{\partial \text{Loss}(\vec{y}, \vec{x}, \vec{W})}{\partial \vec{W}}\bigg|_{\substack{\vec{y}=\vec{y}_k,\\ \vec{x}=\vec{x}_k,\\ \vec{W}=\vec{W}_k+\mu\vec{\Delta}_k}} + \mu\vec{\Delta}_k$$

for some momentum value μ. At the activation function origin, the gradients used in our algorithm are ill-defined and can become large. To prevent overshooting caused by excessively large values for the gradient, in our experiments for this project, we clipped the gradient to the range $[-5, 5]$ where we chose the value 5 through trial and error.

Backpropagation [29] is a technique to extend the SGD algorithm just discussed for single layer networks to multi-layer, artificial neural networks. Suppose we have a layered neural network computing some function $\vec{f}(\vec{x}, \vec{W})$. We write $\vec{W}_i$ to denote the weights used in ith layer of this network. When training we write $\vec{W}_{k,i}$ to denote the weights of the ith layer after k many training items have been considered and set $\vec{\Delta}_{k+1,i} = \vec{W}_{k+1,i} - \vec{W}_{k,i}$. Define $\vec{x_0} = \vec{x}$ as the inputs into a layered neural network, and for $i > 0$, let $\vec{x}_i = \vec{f_i}(\vec{x}_{i-1}, \vec{W}_i)$ denote the function computed by the ith layer of our network as a function of its input lines and weights. Let $\vec{x}_{k,i}$ be the ith layer values after we have trained for k data items. Note we are slightly abusing notation: When we write $\vec{x}_i$ we mean the outputs of the ith layer of our network; whereas, when we write $\vec{x}_k$ we mean the kth training data inputs to our network. We next group the weights in our loss functions from before as

$$\text{Loss}(\vec{y}, \vec{x}, \vec{W}) = \text{Loss}(\vec{y}, \vec{x}, \vec{W}_1, \ldots, \vec{W}_\ell)$$

and in analogy with our update rule for the single layer case define $\vec{W}_{k+1,i}$ as

$$\vec{W}_{k,i} - \lambda \frac{\partial \text{Loss}(\vec{y}, \vec{x}, \vec{W})}{\partial \vec{W}_i}\bigg|_{\substack{\vec{y}=\vec{y}_k,\\ \vec{x}=\vec{x}_k,\\ \vec{W}_i=\vec{W}_{k,i}+\mu\vec{\Delta}_{k,i}}} + \mu\vec{\Delta}_{k,i}.$$

Backpropagation is used to compute $\frac{\partial \text{Loss}(\vec{y},\vec{x},\vec{W})}{\partial \vec{W}_i}$. To do this we use that $\text{Loss}(\vec{y}, \vec{x}, \vec{W}) = E(\vec{y}, \vec{f}(\vec{x}, \vec{W}))$, so by the chain rule, recalling $\vec{x}_i = \vec{f_i}(\vec{x}_{i-1}, \vec{W}_i)$, we have

$$\frac{\partial \text{Loss}(\vec{y}, \vec{x}, \vec{W})}{\partial \vec{W}_i}\bigg|_{\substack{\vec{y}=\vec{y}_k,\\ \vec{x}=\vec{x}_k,\\ \vec{W}_i=\vec{W}_{k,i}+\mu\vec{\Delta}_{k,i}}} = \frac{\partial E}{\partial \vec{x}_i}\bigg|_{\substack{\vec{y}=\vec{y}_k,\\ \vec{x}_i=\vec{x}_{k,i}}} \cdot \frac{\partial \vec{f_i}}{\partial \vec{W}_i}\bigg|_{\substack{\vec{x}_{i-1}=\vec{x}_{k,i-1},\\ \vec{W}_i=\vec{W}_{k,i}+\mu\vec{\Delta}_{k,i}}}.$$

and

$$\frac{\partial E}{\partial \vec{x}_{i-1}}\bigg|_{\substack{\vec{y}=\vec{y}_k,\\ \vec{x}_{i-1}=\vec{x}_{k,i-1}}} = \frac{\partial E}{\partial \vec{x}_i}\bigg|_{\substack{\vec{y}=\vec{y}_k,\\ \vec{x}_i=\vec{x}_{k,i}}} \cdot \frac{\partial \vec{f_i}}{\partial \vec{x}_{i-1}}\bigg|_{\substack{\vec{x}_{i-1}=\vec{x}_{k,i-1},\\ \vec{W}_i=\vec{W}_{k,i}+\mu\vec{\Delta}_{k,i}}}.$$

Given that our neural gates are simple activation functions applied to weighted sums, or some other easy to compute function of its inputs, the derivative $\frac{\partial \vec{f_i}}{\partial \vec{W}_i}$ and $\frac{\partial \vec{f_i}}{\partial \vec{x}_{i-1}}$ are also easy to compute. When $i = \ell$, $\frac{\partial E}{\partial \vec{x}_i}$ will be the same as $\frac{\partial E}{\partial \vec{z}}$ in our single layer case. The second equation above then gives us the necessary recurrence to compute $\frac{\partial E}{\partial \vec{x}_i}$ for layers where $i < \ell$.

Our description of network training so far suffices to handle feedforward networks. The Backpropagation Through Time (BPTT) algorithm [30] extends our algorithm above to handle certain kinds of simple recurrent networks such as those used in this paper. Suppose the inputs to $\vec{f_i}$ is a function of $\vec{x}_{i-1}$, $\vec{x}_{i+m}$, and $\vec{W}_i$, giving a recurrence between the outputs of layer $i + m$ and the inputs to layer i. Assume also that there no other recurrences among the layers from i to $i + m$. Assume if there are other disjoint layers with recurrences they involve fewer than m layers, and we carry out a similar procedure on them as that which we are about to describe. Let $\vec{g}(\vec{x}_{i-1}, \vec{x}_{i+m}, \vec{W}_i')$ be the composition of the layers involved in the recurrence starting with $\vec{f_i}$ where $\vec{W}_i' = \vec{W}_i, \ldots, \vec{W}_{i+m}$. Rather than training on single example, $(\vec{x}_k, \vec{y}_k)$ at the kth training step, in BPTT, we train on t input examples and one output example, written as $(\vec{x}_k, \vec{x}_{k+1}, \ldots, \vec{x}_{k+t-1}, \vec{y}_{k+t-1})$, together with a randomly initialized $\vec{z}_0$. The m layers to compute g are replace with $t \times m$ layers computing

$$\vec{z}_1 = g(\vec{x}_{k,i}, \vec{z_0}), \vec{z}_2 = g(\vec{x}_{k+1,i}, \vec{z_1}), \ldots, \vec{z}_t = g(\vec{x}_{k+t-1,i}, \vec{z}_{t-1})$$

where $\vec{z}_t$ is then fed in as inputs into $i + m$th layer of our network. This t-wise unfolding of g is used to approximate the behavior of the recurrence, but the modified network is now feedforward so can be trained with backpropagation.

Notice in our description above of how we combined the original data pairs into a vector to present to our recurrent network, we combined the data items in a forward direction: $\vec{x}_{i-1}, \vec{x}_i, \ldots, \vec{x}_{i+m}$. It is also possible to present the training sequence in a backwards direction or to train on both forwards and backwards directions (bidirectional). In the literature, all three possibilities have been considered [5]. Although we tried different variants, for this report we are presenting results for forward only BPTT training which seemed to give the best results for our problem.

B. Neural Network Layer Types

In this subsection, we describe the various kind of neural network layers we use to build up our CAPTCHA-breaker. Again, as we have said earlier in this paper, we are choosing specific types of layer topologies geared toward the task at hand to reduce the number of weights that need to be trained for.

The first kind of neural network layer we consider is a **convolutional layer**. Convolutional layers were first introduced by Yann LeCun, et al. [14] as part of their convolutional neural network (CNN) system to perform handwritten character recognition. The inputs to a convolutional layer are assumed to be data of some fixed dimensionality, in our case two dimensional arrays derived from image data from one or more sources, the outputs are

one or more **feature maps**. For the rest of this discussion, we assume we are dealing with two dimensional data, but the concept can easily be generalized to a k-dimensional setting, with more tweakable parameters than we need. For us, each feature map has two tunable parameters: a filter shape, which is a $t \times t$-sized region, and a bias. If an input consists of $n \times m$ arrays of real data from k many sources, a feature map will contain $(n-t+1) \times (m-t+1)$ neurons each of which will use the same $t \times t \times k$+1 weights. The '+1' is the bias weight. The inputs to the (i,j) neuron of the feature come from each source $1,\ldots,k$ from the rectangle with top left (i,j) and bottom right $(i+t, j+t)$ in the inputs. The weighted sum of these inputs with the bias added is fed into the activation function to compute the value of such a neuron. Intuitively, one feature map allows us to train for a $t \times t$-sized feature that appears somewhere at the same locations in the original images. Neurons which output the largest values in the feature map correspond to appearances of that $t \times t$ feature in the original images, allowing us to find any translations of that feature. If we have multiple convolutional layers, then lower layers learn low-level features and higher layers learn high-level features. In particular, a first layer might have feature maps that take care of detecting edges and use just one source, and following layers might have feature maps that use multiple sources but can detect different shapes built out of edges.

When building a neural network to recognize shapes, convolutional layers are often immediately followed by so-called maxpool layers. A **maxpool layer** consists of a set of max units each taking $r \times r$ many inputs. A max unit computes the maximum value among these inputs. When used together with a convolutional layer, the output lines from each $s \times s$ dimensional feature map are divided into $\lceil s/r \rceil \times \lceil s/r \rceil$ many non-overlapping rectangles and each rectangle is fed into a max unit. If s is not divisible by r than some of the max-units might have some of their inputs padded by 0's. If one did not use maxpool layers, and one had a network with several convolutional layers, than the number of weights could grow quickly unmanageable. To see this just note that $n \times m$ inputs in a first convolutional layer might result in many features maps of close to the same size. Then in a second layer each of those feature map outputs could serve as inputs to several more features maps, and so on. Having a maxpool layer after each convolutional layer reduces the sizes of the features maps in deeper layers preventing the number of weights that need to trained from growing too unmanageable. Having a maxpool also tends to prevent features from being trained too exactly to the input data, that is, features which are overfitted to the inputs. For example, suppose we had two features related to parts of a character extracted in a first convolutional layer. A maxpool layer would reduce the importance of the exact positions of these features relative to each other when these features are fed into an additional convolutional layer that is extracting something closer to whole character features.

After several higher-order features have been extracted from image data it is often useful to have a fully connected layer, a dense layer, to synthesize these features. A **dense layer** consist of units each of which takes as inputs all the unit outputs of the previous layer. Since a dense layer might have many weights that need to be trained, it is prone to overfitting. In order to control for this, the drop out method is used: At each training step, nodes are dropped out of this layer with a probability p leaving a reduced network with fewer weights. During a given training step, the training is done with respect to this reduced network. After the step is over, the "dropped" nodes are reinserted in the network, and the process is repeated for the next training data item. This helps in preventing units co-adapting, and hence, helps in overcoming overfitting. For our results, we used a drop out probability of 0.5.

To handle variable length CAPTCHAs our network models will make use of an **LSTM (Long Short Term Memory) layer**, a type of recurrent neural network (RNN) layer. Units in an LSTM layer are built out of several subcomponents. LSTMs were first defined as in [12] [13]. These versions did not have forget gates and peephole connections common today in the literature. Greff, et al. [7] has a good survey and analysis of LSTM variants. The version of LSTM unit used for our results is based on Graves [6]. The recurrent aspect of these units means that one talks about values of subcomponents at a given time step t. At the heart of an LSTM unit is a vector-valued memory cell which is useful to exploit long range dependencies in the data. Let $\vec{c}_t$ denote the memory cell state of the unit we are describing at time t. Similarly, let $\vec{x}_t$ denote the inputs to the LSTM unit at time t and $\vec{y}_t$ its outputs. Three internal gate vectors control how the memory cell is updated and the influence of the memory cell and the inputs on the output values of the LSTM unit: the input gate vector $\vec{i}_t$, the forget gate vector $\vec{f}_t$, and the output gate vector $\vec{o}_t$. The rough idea of how the gates operate is: If $\vec{f}_t$ is close to $\vec{0}$, then in the next time step the memory cell will have value close to zero. The degree to which the inputs $\vec{x}_t$ get written into the memory cell in the next time step is otherwise controlled by the input gates value $\vec{i}_t$. Similarly, the output gate vector, $\vec{o}_t$, and the current memory cell vector, $\vec{c}_t$, are used to compute the output. As we will see, $\vec{f}_t$, $\vec{i}_t$, $\vec{o}_t$ are dependent $\vec{y}_{t-1}$ and so the unit will be recurrent.

To give the formal definition of how the vectors above are computed from inputs $\vec{x}_t$ and prior values, we need to introduce some notation. LSTMs make use of two different activation functions: the sigmoid function $g(x) = (1+e^{-x})^{-1}$ and $tanh(x) = \frac{e^x - e^{-x}}{e^x - e^{-x}}$. Given a k-dimensional vector $\vec{u}$, we write $\vec{g}(\vec{u})$, $\vec{tanh}(\vec{u})$ to denote the component-wise application of these functions. So is $\vec{g}(\vec{u}) = \langle g(u_1), \ldots, g(u_k) \rangle$. We write $\vec{u} \odot \vec{v}$ for $\langle u_1 \cdot v_1, \ldots u_k \cdot v_k \rangle$. Given these preliminaries the vectors $\vec{i}_t$, $\vec{f}_t$, $\vec{o}_t$, and $\vec{y}_t$ are defined as:

$$
\begin{aligned}
\vec{i}_t &= \sigma_i(\vec{W}_{xi}\vec{x}_t + \vec{W}_{yi}\vec{y}_{t-1} + \vec{W}_{ci} \odot \vec{c}_{t-1} + \vec{b}_i) \\
\vec{f}_t &= g(\vec{W}_{xf}x_t + W_{yf}\vec{y}_{t-1} + \vec{W}_{cf} \odot \vec{c}_{t-1} + \vec{b}_f) \\
\vec{c}_t &= \vec{f}_t \odot \vec{c}_{t-1} + \vec{i}_t \odot \vec{tanh}(\vec{W}_{xc}\vec{x}_t + \vec{W}_{yc}\vec{y}_{t-1} + \vec{b}_c) \\
\vec{o}_t &= g(\vec{W}_{ox}\vec{x}_t + \vec{W}_{oy}\vec{y}_{t-1} + \vec{W}_{oc}\vec{c}_t + \vec{b}_o) \\
\vec{y}_t &= \vec{o}_t \odot \vec{tanh}(\vec{c}_t).
\end{aligned}
$$

In the above, $\vec{b}_i$, $\vec{b}_o$, $\vec{b}_c$ are trainable bias weights, and matrices of the form $\vec{W}_{-i}$, $\vec{W}_{-o}$, $\vec{W}_{-c}$, $\vec{W}_{-f}$ where '-' is x,y, or c

are trainable weight matrices for the input, forget, cell, output gates for the respective vectors $\vec{x}$, $\vec{y}$, or $\vec{c}$.

As we mentioned before, LSTMs are recurrent layers, and so are trained using BPTT, and this can be done in either a forward, backward, or bidirectional fashion. To develop some intuition about how the memory cell might be used, we look at an example from speech recognition. Consider a sentence like: "I __ a girl" that a network wants to complete. In this case, we can make use the word "I" to predict that the next word should be am. So having a memory of the previous word "I" could be useful in predicting the next word. A forward LSTMs first predicts "I, and then "am, and so on. Backward LSTMs predict backwards. For example, in the above example, it will predict "girl" first, then "a", then "am" and lastly "I". In bidirectional LSTMs, both forward and backward LSTMs training is combined. An LSTM trained this way tries to predict given both the forward and backward contexts. For example: "She lives in __. She speaks French." In this sentence, we know that because of "lives", we need to fill the name of a place. But by only looking at French, could we make a prediction that it would be France. In these scenarios, bidirectional LSTMs are useful. We also expect bidirectionally-trained LSTMs to be useful in the CAPTCHA recognition problem. For instance, in problems like "rrn", the output could be ['r', 'm'] or ['r', 'r', 'n'], so it is useful to predict if the last character is 'm' or 'n' first and then predict the previous characters.

The last kind of layer used in our artificial neural network models is a softmax layer. A **softmax layer** with k inputs and m outputs has m units each of which compute weighted sums of all k inputs, the m outputted sums $s_1, \ldots, s_m$ are then fed into the softmax function which computes:

$$\vec{f}(\vec{s}) = \left\langle \frac{e^{s_1}}{\sum_{i=1}^{m} e^{s_i}}, \frac{e^{s_2}}{\sum_{i=1}^{m} e^{s_i}}, \ldots, \frac{e^{s_m}}{\sum_{i=1}^{m} e^{s_i}} \right\rangle$$

I.e., the coordinates of $\vec{f}$ are the m outputs of the softmax layer. Notice the softmax layer outputs will all be in the range $[0, 1]$ and the sum of these outputs will be 1, so these outputs can be viewed as a probability distribution over the classes represented by each output line.

V. Dataset

Even using the various kinds of layers of the last sections, which were specifically developed to reduce the number of weights needed to train neural network models, the number of weights we need to train to handle CAPTCHA breaking is still large. So we needed a large dataset of CAPTCHAs to train these weights. A standard dataset for CAPTCHAs is not publicly available, so we developed one prior to conducting our CAPTCHA training experiments. We used the BSD-licensed, Simple CAPTCHA Java library [24] to create our CAPTCHAs. It has various functions to create noise and backgrounds in a CAPTCHA image. We wrote a Java module that generates a random string of 4 to 7 characters in length and then randomly adds noise to this image. The background of an image was chosen with the same randomness. Figure 2 and Figure 3 are example simple and complex CAPTCHA images we generated with our program.

Fig. 2 A simple CAPTCHA example

Fig. 3 A complex CAPTCHA example

All the images generated are of same size, 200×50 pixels. We trained our model using simple CAPTCHA images, complex CAPTCHA images, and a combination of both. Simple images are those that have very little noise, as shown in Figure 2, and complex images contain more noise and clutter, as shown in Figure 3. We created four datasets: a set consisting of 1 million simple images of fixed length 5, a set consisting of 2 million complex images of fixed length 5, a set consisting of 13 million images of variable length chosen with equal frequency to use the simple or complex CAPTCHA generation, and a set of a million "live" images based on a CAPTCHA system found on the web. We used the text that was contained in the CAPTCHA image as part of its filename when saving it to disk. The ordered pair, (image, text from filename), then served as a training data item when we did neural network training. Before supplying images for training, we converted them to grayscale using a PIL library [22] function to reduce a RGB color images to grayscale image via the following formula:

$$L = R \cdot 299/1000 + G \cdot 587/1000 + B \cdot 114/1000.$$

VI. Our Models

There are many publicly available frameworks to train artificial neural network models. Some of them are: Torch7, Caffe, and Theano. We chose Theano because we found Theano to be more flexible and because it has GPU support. At its core, Theano is a numerical computation library written in Python for multi-dimensional arrays. An additional library, Lasagne, lets one easily define neural network layers as Theano expressions which can then be trained using Theano function calls. Since Theano is written in Python, we wrote our system in Python to easily make use of it. We created two kinds of CAPCTHA-breaking artificial neural network models: a multiple Softmax layer model and an LSTM-based model. For the LSTM model, we trained on both fixed length and variable length CAPTCHAs. The code for all of our models as well as the code to generate our datasets is available on GitHub at https://github.com/bgeetika/CAPTCHA-Decoder. We also created a website to perform CAPTCHA breaking online at http://cp-training.appspot.com/ .

At the bottom of all of our models, we had 200×50 inputs which were intended to come from grayscale image pixels,

where each pixel represented a nonnegative float. The first layer of all of our models is a convolutional layer that takes these inputs and produces 32 feature maps. Each feature map is computed based on a 5×5-sized filter. After this convolutional layer, we had a maxpool layer which used a 2×2 matrix to get the max of neighboring pixel values. The next layer was again a convolutional layer. This layer also outputs 32 feature maps, but in this case the feature maps used 5×5-sized filters and all 32 feature map output from the previous maxpool layer as sources. This second convolutional layer was again followed by 2×2 matrix maxpool layer. The outputs from this second maxpool layer correspond to 32 two dimensional array of size 50×12. The remainder of the network depended on which particular model was being considered.

Figure 4 shows our multiple softmax model. In it the outputs from the last maxpool layer just described are fed to five parallel dense layers each made up of 256 units. The 256 outputs from each of these five layers are then each sent to a softmax layer with 256 inputs and 62 outputs. Here the number 62 is based on the number of character classes we are training for: 26 upper case characters, 26 lower case characters, 10 digit characters. This model was trained using a sequence of tuples

$$(\vec{x}_1, \vec{y}_{1,1}, \vec{y}_{1,2}, \ldots, \vec{y}_{1,5}), (\vec{x}_2, \vec{y}_{2,1}, \vec{y}_{2,2}, \ldots, \vec{y}_{2,5}), \ldots$$

where $\vec{x}_i$ was one 200×50 pixel CAPTCHA from one of our datasets, and $\vec{y}_{i,1}, \vec{y}_{i,2}, \ldots, \vec{y}_{i,5}$ were five 62-dimensional vectors with only 1 coordinate set to 1. The coordinate position set to 1 in $\vec{y}_{i,j}$ indicates the character that was supposed to be at the jth position in that CAPTCHA. It is an all 0 vector except for a 1 in the coordinate corresponding to the desired character class. When testing our trained model, a CAPTCHA image was fed in as inputs and the five length 256 vectors were computed as outputs. Each vector was then decoded into a character based on which coordinate in that vector had the largest value. Notice in this model, and this will also be in the case in our later models, each character is computed based on the whole image.

The second model we considered was a fixed-length CAPTCHA for training, LSTM model. In this model, the outputs of the final maxpool layer were fed to a dense layer with 256 units. All the outputs of these units are then each fed into 256 LSTM units. The 256 outputs of these units are then fed into a softmax layer. At training time, we used a BPTT to train this network using a 5-way unfolding in a forward fashion of the LSTM layer as illustrated by Figure 5. The unfoldings in this figure are shown horizontally. A training tuple was again of the form $(\vec{x}_i, \vec{y}_{i,1}, \vec{y}_{i,2}, \ldots, \vec{y}_{i,5})$ with the $\vec{x}_i$ image data fed as inputs and the expected character vectors $\vec{y}_{i,j}$ used as outputs in the unfolded network during backpropagation. At testing time, an input image was fed into the network, and the output values of the network were computed for five steps through the recurrent LSTM layer. Each output was decoded as we did above.

To modify our second model so that it could handle variable length CAPTCHAs, we introduced a new possible output vector code, 'stop', in addition to our codes for upper case, lower case, and digit characters. This code was used to

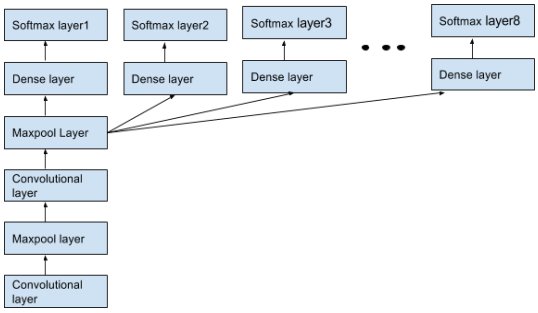

Fig. 4 CNN with multiple softmax

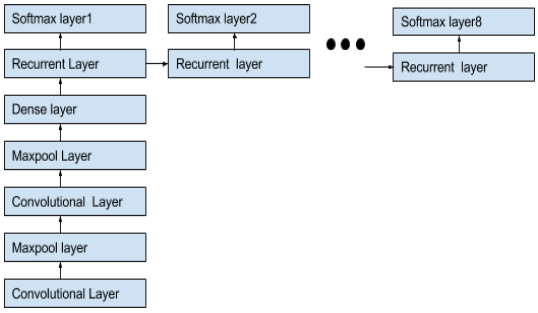

Fig. 5 RNN architecture

represent that the CAPTCHA string was done. I.e., that it had no more characters. To test how well this worked, we trained our network with a 7-wise unfolding of the LSTM layer on fixed-length length 5 CAPTCHAs. A training tuple in this setting was of the form

$$(\vec{x}_i, \vec{y}_{i,1}, \vec{y}_{i,2}, \ldots, \vec{y}_{i,5}, \text{stop}, \text{stop}).$$

As our initial tests seemed successful, we then trained our model on CAPTCHAs of randomly chosen lengths from four to seven characters using the same idea of the stop vector to indicate in a training tuple when the CAPTCHA was done.

VII. Experiments and Results

We used the datasets described in Section V to train the neural network models just described. To make training our models more efficient, we packaged 20,000 images at a time into single numpy files. We coded our training script so that one could specify this batch size. Our results are given in Table I and Table II.

Type of model	Character Accuracy
Multiple Softmax fixed length(Simple dataset)	99.8%
Multiple Softmax fixed length(Complex dataset)	98.96%
LSTM fixed length (Simple Dataset)	100%
LSTM fixed length (Complex dataset)	98.48%
LSTM with "live" data	99.2%
LSTM variable length with fixed length data	99.5%
LSTM variable length with variable length data	97.31%

TABLE I Individual character accuracy for different models

VIII. Conclusion

In this paper, we have described our experiments in trying to decode character-based image CAPTCHAs using deep neural

Type of model	Sequence Accuracy
Multiple Softmax fixed length(Simple dataset)	99%
Multiple Softmax fixed length(Complex dataset)	96%
LSTM fixed length (Simple Dataset)	99.8%
LSTM fixed length (Complex dataset)	91%
LSTM with "live" data	97%
LSTM variable length with fixed length data	98%
LSTM variable length with variable length data	81%

TABLE II Likelihood whole captcha decoded correctly for different models

networks. We used convolutional layers and recurrent layers instead of using the conventional approach of first cleaning a CAPTCHA image, segmenting the image, and recognizing the individual characters. For machine learning problems, we need a large amount of data, so we have generated a dataset of 13 million character-based image CAPTCHAs. Our models were trained on both simple CAPTCHAs and complex CAPTCHAs. The accuracy achieved on fixed-length CAPTCHAs was impressive (99% for simple images and 96% for complex images) for the multiple softmax model. We tried both fixed length and variable length CAPTCHAs. These gave 99% and 81% accuracies respectively. Gradient clipping helped speed up the training of our LSTMs, which was otherwise very slow. It is clear that the more kinds of CAPTCHAs we include in our training set, the more robust our model will become. We have tried to demonstrate this by using a real dataset in our training set, and were able to achieve 99% accuracy. While it is still in early stage, our model performs better than previous work that relies on manually-generated segmentation oriented models. For example, our model beats the accuracy of [19]'s model using comparably-hard CAPTCHA generator.

REFERENCES

[1] L. von Ahn, M. Blum. N. J. Hopper, and J. Langford. CAPTCHA: Using Hard AI Problems for Security. *EUROCRYPT 2003: International Conference on the Theory and Applications of Cryptographic Techniques*. 2003. pp. 294–311.

[2] M. D. Lillibridge, M. Abadi, K. Bharat, A. Z. Broder. U.S. Patent 6,195,698. Method for selectively restricting access to computer systems. Filed on Apr 13, 1998 and granted on Feb 27, 2001.

[3] K. Chellapilla and P. Y. Simard. Using Machine Learning to Break Visual Human Interaction Proofs (HIPs). *Advances in Neural Information Processing Systems.* Volume 17. NIPS 2004. pp. 265–272. 2004.

[4] S. Dehaene. *Reading in the Brain*. Penguin Books. 2010.

[5] A. Graves and J. Schmidhuber. Framewise phoneme classification with bidirectional LSTM and other neural network architectures. *Neural Networks*. Volume 18. Issues 5–6. pp. 606–610. July 2005.

[6] A. Graves. Generating sequences with recurrent neural networks. *The Computing Research Repository*. CoRR abs/1308.0850. 2013.

[7] K. Greff, R. K. Srivastava, J. Koutník, B. R. Steunebrink, and J. Schmidhuber. LSTM: A Search Space Odyssey. *The Computing Research Repository*. CoRR abs/1503.04069. 2015.

[8] I. J. Goodfellow, Y. Bulatov, J. Ibarz, S. Arnoud, V. Shet. Multi-digit Number Recognition from Street View Imagery using Deep Convolutional Neural Networks. *International Conference on Learning Representations 2014*. arXiv:1312.6082. 2014.

[9] A. Hannun, C. Case, J. Casper, B. Catanzaro, G. Diamos, E. Elsen, R. Prenger, S. Satheesh, S. Sengupta, A. Coates, A. Y. Ng. Deep Speech: Scaling up end-to-end speech recognition. *The Computing Resource Repository*. CoRR abs/1412.556717. 2014.

[10] S. O. Haykin. *Neural Networks and Learning Machines*. 3rd Ed. Pearson. 2011.

[11] S. Hochreiter. Untersuchungen zu dynamischen neuronalen Netzen. *Long short term memory.* Technische Univ. Munich, 1991.

[12] S. Hochreiter and J. Schmidhuber. Long Short Term Memory. Technical Report FKI-207-95. Technische Universität Mu nchen, München. August 1995.

[13] S. Hochreiter and J. Schmidhuber. Long Short Term Memory. *Neural Computation*. Volume 9. Issue 8. pp 1735–1780. November 1997.

[14] Y. LeCun, B. Boser, J. S. Denke, D. Henderson, R. E. Howard, W. Hubbard and L. D. Jackel. Handwritten digit recognition with a back-propagation network. *Advances in Neural Information Processing Systems.* Volume 2. NIPS 1989. pp. 396–404. 1989.

[15] S. McCulloch and W. H. Pitts. Resource Description for A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*. Volume 5. pp. 115–133. 1943.

[16] G. W. McConkie, and K. Rayner. The Span of the Effective Stimulus during a Fixation in Reading. em Perception and Psychophysics. Volume 17. pp. 578–586. 1975.

[17] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean. Distributed Representations of Words and Phrases and their Compositionality. *Advances in Neural Information Processing Systems*. Volume 26. NIPS 2013. pp. 3111–3119. 2013.

[18] M. S. Minsky and S. Papert. em Perceptrons: An Introduction to Computational Geometry. MIT Press, 1969.

[19] G. Mori and J. Malik. Recognizing Objects in Adversarial Clutter: Breaking a Visual CAPTCHA. *IEEE Conference on Computer Vision and Pattern Recognition*. CVPR 2003. Vol. 1. IEEE Computer Society. pp.134–141. 2003.

[20] M. Naor. Verification of a Human in the Loop, or Identification via the Turing Test. Unpublished Manuscript. 1996.

[21] G. Templeton. Artificial neural networks are changing the world. What are they? http://www.extremetech.com/extreme/215170-artificial-neural-networks-are-changing-the-world-what-are-they. Extremetech.com. Oct.12, 2015.

[22] Python Imaging Library. Retrieved June 2016 from http://www.pythonware.com/products/pil/.

[23] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning internal representations by error propagation," in em Parallel Distributed Processing: Explorations in the Microstructure of Cognition. Volume 1. Bradford Books. pp. 318–362. 1986.

[24] SimpleCAPTCHA. Retrieved June, 2016 from http://simplecaptcha.sourceforge.net/.

[25] I. Sutskever, J. Martens, G. E. Dahl, G. E. Hinton. On the importance of initialization and momentum in deep learning. *Proceedings of the 30th*

International Conference on Machine Learning. ICML 2013. Volume 3. pp. 1139–1147. 2013.

[26] I. Sutskever, O. Vinyals, and Q, V. Le. Sequence to Sequence Learning with Neural Networks. *Advances in Neural Information Processing Systems*. Volume 27. NIPS 2014. pp. 3104–3112. 2014.

[27] Y. Taigman, M. Yang, M. A. Ranzato, and L. Wolf. DeepFace: Closing the Gap to Human-Level Performance in Face Verification. *IEEE Conference on Computer Vision and Pattern Recognition*. CVPR 2014. IEEE Computer Society. pp. 1701–1708. 2014.

[28] O. Vinyals, A. Toshev, S. Bengio, and D. Erhan. Show and Tell: A Neural Image Caption Generator. *IEEE Conference on Computer Vision and Pattern Recognition*. pp. 3156–3164. 2015.

[29] P. J. Werbos, Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Science. em PhD thesis. Harvard University.1974.

[30] P. J. Werbos (1988). Generalization of backpropagation with application to a recurrent gas market model. *Neural Networks* Volume 1 Issue 4. pp. 339–356. 1988.