Technische Universität München

Department of Mathematics

Master's Thesis

Multi-digit Character Recognition of `CAPTCHAs` with Deep Neural Networks

Fabian Stark

Supervisor:	Prof. Dr. Daniel Cremers
Advisors:	M.Sc. Caner Hazırbaş
	Dr. Rudolph Triebel
Date:	11/06/2015

I hereby declare that this thesis is my own work and that no other sources have been used except those clearly indicated and referenced.

Garching, June 11, 2015 Fabian Stark

Abstract

CAPTCHAs are automated tests to tell computers and humans apart. They are designed to be easily solvable by humans, but unsolvable by machines. We automatically solve these tests with Convolutional Neural Networks (CNNs) in this thesis. We therefore combine the localization, segmentation and recognition of multi-character text in one network, because in modern CAPTCHAs the characters overlap each other and thus cannot be segmented any more with rectangular windows. We apply our method on easy undistorted sequences and also on complicated distorted CAPTCHAs. However, the strength of CNNs relies on the training data that the classifier is learnt on and especially on the size of the training set. Hence, it is intractable to solve the problem with CNNs in case of insufficient training data. We propose an Active Deep Learning strategy that makes use of the ability to gain new training data for free without any human intervention which is possible in the special case of CAPTCHAs. We discuss how to choose the new samples to re-train the network and present results on an auto-generated CAPTCHA dataset. Our approach dramatically improves the performance of the network if we initially have only few labeled training data.

Contents

Part I.

Introduction and Theory

1. Introduction

A CAPTCHA [1] (**C**ompletely **A**utomated **P**ublic **T**uring test to tell **C**omputers and **H**umans **A**part) is an automated test to identify whether the user is a human or not. These tests are widely used on the internet as a security measure to prevent automated programs from abusing online services. Nowadays, in almost all web services users have to prove that they are humans by solving CAPTCHAs. Examples include the registration for an email account or other web services, as well as online shopping. The most commonly used CAPTCHAs require the user to type the letters of a distorted text. For humans this is most often a simple task, while computers still have difficulties to solve CAPTCHAs. Ideally, a human should solve such a Human Interaction Proof in more than 80% of the time, while an automatic script with reasonable resource use should succeed less than 0.01% of the time [2]. One Example of CAPTCHAs is shown in Figure 1.1.

Besides websites that offer an API to upload the image of a CAPTCHA and get it solved by another human[1], researchers have recently started investigating automated methods to solve CAPTCHAs. Many of these existing solutions first perform character segmentation and then recognition. However, they cannot solve newer, more challenging CAPTCHAs, where the letters are skewed so that they cannot be separated by vertical lines. Thus, rectangular windows cannot be used for segmentation, and more powerful classification methods are needed. One successful approach proposed recently by Goodfellow *et al.* [3] uses a deep Convolutional Neural Network (CNN), a framework that is also used in many other tasks such as object classification [4, 5], automatic speech recognition [6, 7] or natural language processing [8, 9]. However, a major requirement for training a deep CNN is a very large training data set (for example the ImageNet [10] data for image classification), and for CAPTCHA recognition, there is usually only a small annotated training data set available. Furthermore, the appearance of CAPTCHAs can, in contrast to objects in natural images, often change significantly from those available in the training data, for example due to a major change in the distortion applied to the text.

To address these problems, we also propose an approach that is based on Active Learning in this thesis. The idea here is to start learning with a comparably small training set and add new training samples in every subsequent learning round. The decision whether a sample is added to the training data is based on the uncertainty that the classifier associates with a given prediction. Under the assumption that this uncertainty estimation is well calibrated, the algorithm selects the most informative samples to learn from, resulting in less training samples required than in standard passive learning. As a further advantage, the algorithm can adapt to new input data that differ in appearance from the current training data. We note however that our problem is different to other Active Learning

[1]for example: www.9kw.eu promisses to solve nearly every CAPTCHA in less than 30 seconds. In return the user has to solve CAPTCHAs for other users when he has time, or he can pay some money.

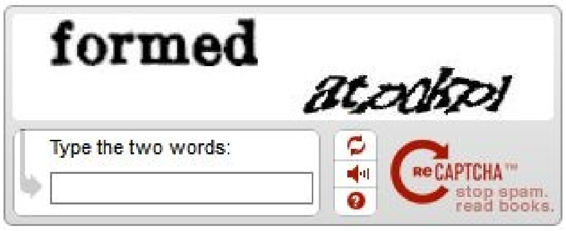


Figure 1.1.: Google's reCAPTCHA [1]. It consists of one distorted sequence (right) and one scanned word from a text book (left), and the user is asked to type both words, thereby helping to digitize printed books.

settings in that we do not need a human supervisor to acquire the ground truth labels for training. Instead, we use the return value that we obtain automatically when solving a CAPTCHA. Thus, if the classifier is able to solve a CAPTCHA correctly we can use that information for re-training, because then the ground truth label is known. Of course, in case the CAPTCHA is not solved we don't have that information, but we will show how learning can be done from the correctly predicted samples only.

1.1. Related Work

Conventional methods aim at detecting the text within natural images in two disjoint steps as described in [11]: localizing the regions of words, or single characters within the image, segmenting [12] and then recognizing them [13]. In addition, a dictionary can be used to dismiss unlikely words. For example, Mori and Malik [14] proposed a method to solve CAPTCHAs using a dictionary. They only store 411 words in the dictionary, because the CAPTCHAs they solve only contain those words. In [2] even harder CAPTCHAs are solved by first segmenting the single characters of the displayed text and then recognizing them, but without using any dictionary.
However, in modern CAPTCHAs single characters cannot be segmented any more with rectangular windows, because the characters overlap each other. One example is shown in Figure 1.1 (right hand side) and more examples can also be found in Figure 5.1b. These CAPTCHAs are more similar to hand-written text, and LeCun *et al.* [15] proposed to use Convolutional Neural Networks for recognition of hand-written digits. These networks are designed to construct the hierarchy of the objects layer by layer and perform classification. In 2014 Goodfellow *et al.* [3] proposed a method that combines the localization, segmentation and recognition of multi-character text via the use of deep Convolutional Neural Networks. They report an accuracy of 99.8% for solving Google's hardest reCAPTCHAs which is better than every human. In Figure 1.2 there are two examples of those hardest reCAPTCHAs which are really even for humans difficult to read. For solving the CAPTCHAs they use a network with 9 convolutional layers and 2 fully connected layers with a training set in the order of millions of images. To train the network they are using the DistBelief implementation [16] on a cluster of several computers and still need some days, so it is not possible to train the net on commodity hardware.

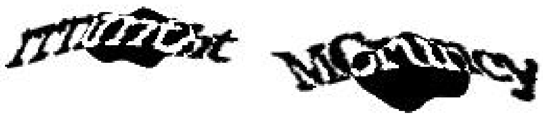


Figure 1.2.: Two examples of Google's hardest reCAPTCHAs [3].

As already mentioned (compare Figure 1.1), the reCAPTCHA system of Google displays one distorted computer generated string and one with a word from a scanned document that OCR software has been unable to read. In [17] a machine learning algorithm is presented to tell the difference between "control word" and "unknown word" with an accuracy of 99.83%. The "control word" can be solved as described above and for the "unknown word" some random string can be submitted as the system does not know the answer itself. Thus the system of reCAPTCHA could be compromised and lose the ability of leveraging human mental efforts to help digitizing texts.

1.2. Contributions

In this thesis we combine the localization, segmentation as well as the recognition of text in CAPTCHAs in one Convolutional Network that operates directly on the image pixels and detects the whole sequence at once without prior segmentation. We train the network with the Caffe [18] deep learning framework on commodity hardware. We start with applying our method on easy undistorted sequences, but also apply it on complicated distorted CAPTCHAs later.
Furthermore, we show how to reduce the size of the training set by exploiting Active Learning in order to fine-tune the network during runtime. Our network is fed with correctly classified but highly uncertain test samples. Therefore we first show how to compute uncertainty from a deep CNN and how this relates to correct classification. Second, we perform Active Learning with a deep CNN. And third, we show that already the correct, but uncertain classified samples are enough for efficient learning, with the effect that we need only little initial training data, and this is obtained without any human intervention.

2. Theory

The difficulty of visual pattern recognition becomes apparent if we attempt to write a computer program to recognize digits. What seems easy when we do it ourselves suddenly becomes extremely difficult. Simple intuitions about how we recognize shapes (like a 9 has a loop at the top, and a vertical stroke in the bottom right) turn out to be not so simple to express algorithmically. When we try to make such rules precise, we quickly get lost in a morass of exceptions and caveats and special cases. It seems hopeless. [19]

Neural networks approach the problem in a different way. The idea is to take a large number of training examples, and then develop a system which can learn from them. One differentiates between supervised learning which means that we tell the system the correct labels of the training data and unsupervised learning where the system does not know any labels.
In this thesis we purely use supervised learning. For unsupervised training, the network tries to extract features from which it can rebuild the original image. More details can be found in [20].

In this chapter we first of all want to have a closer look at neural networks in Section 2.1. Afterwards we want to explain the special case of Convolutional Neural Networks in Section 2.2.

2.1. Neural Networks

Neural networks and deep learning are the best solutions to many problems in image recognition at the moment. In this section we will provide the basic concepts behind neural networks and deep learning. This section is adapted from [19] where a much more detailed introduction to neural networks can be found.

2.1.1. Basics

A neural networks is a statistical learning algorithms inspired by biological neural networks (for example the brain) in order to approximate a function that can depend on a large number of inputs. It is easily possible to estimate very complicated functions. In Figure 2.1 there is an example of a very easy neural network with 5 inputs and 1 output.

The circles represent interconnected *neurons*, 3 in the first so called hidden layer, 5 in the second and 1 in the last. A neuron takes several inputs and calculates an output. In Figure 2.2 we can see a neuron with 3 inputs.

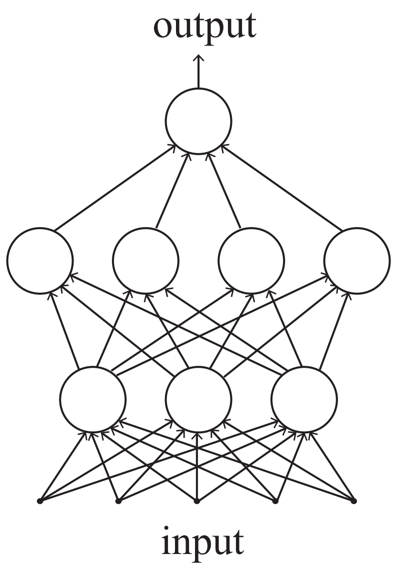


Figure 2.1.: Example of an easy neural network.

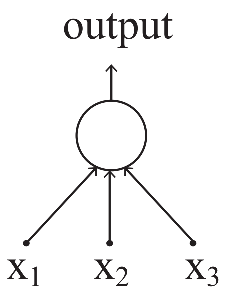


Figure 2.2.: Example of a neuron with 3 inputs.

One easy example of a neuron is a *perceptron* which only takes binary inputs and also has a binary output which is computed as follows:

$$\text{output} = \begin{cases} 0, & \text{if } \sum_j w_j x_j + b \leq 0, \\ 1, & \text{if } \sum_j w_j x_j + b > 0 \end{cases} . \tag{2.1}$$

The so called *weights* (denoted by w_j) and *bias* (denoted by b) for every neuron are learned automatically by showing the network inputs for which we know the desired output. After training the network with those *training images*, it can afterwards also predict the output for other inputs that it has not explicitly learned. So for example it is possible to train a neural network to recognize handwritten digits (the raw pixel data from a scanned, handwritten image of a digit are the input to the network and the predicted digit is the output).

To emphasis the strength of neural networks, we want to cite the following statement: It is possible to use networks of perceptrons to compute any logical function at all and it follows that perceptrons are also universal for computation. [19]

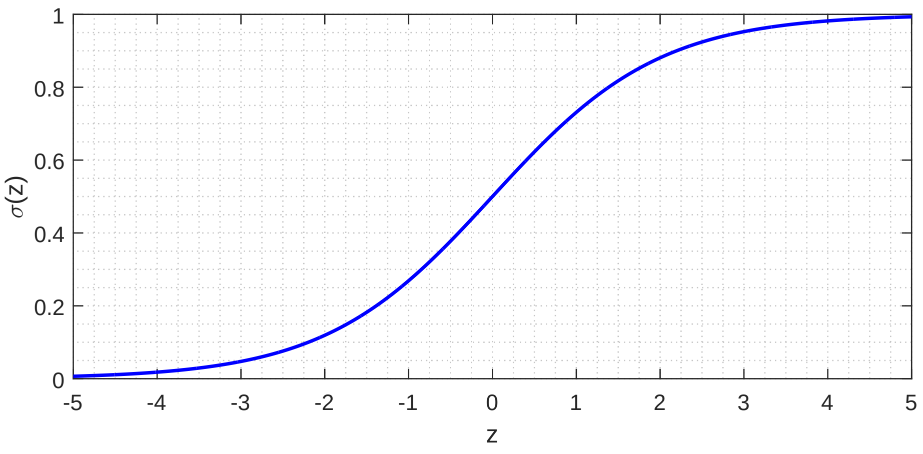


Figure 2.3.: Shape of the sigmoid function.

But in order to train a network, the discrete perceptrons are not perfect, so we want to introduce *sigmoid neurons*. Sigmoid neurons cannot only take binary inputs, but any input between 0 and 1 and they also output a value between 0 and 1 instead of a binary output. So instead of equation 2.1, the output is computed by

$$\sigma(wx + b) \tag{2.2}$$

where σ is the *sigmoid function*:

$$\sigma(z) = \frac{1}{1 + e^{-z}}. \tag{2.3}$$

We can see the graph of the sigmoid function in Figure 2.3.

The shape of the sigmoid function is a smoothed out version of the step function. The step function is plotted in Figure 2.4. If we would replace the sigmoid function by the step function, we would get back our perceptron.

The smoothness of the σ function is the crucial fact why it is much better than the step function for training neural networks.

Sometimes we do not want to use the sigmoid function, but an activation function which is defined as

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

This function is called *rectified linear unit* (ReLU) function. It has been argued to be more biologically plausible than the sigmoid function. Note that this function has a codomain of $[0, \infty)$ whereas in contras the sigmoid function has a codomain of $(0, 1)$.

2.1.2. Optimization Problem

What we now want our algorithm to do is to find weights and biases such that the output from the network approximates for all training inputs x the output $y(x)$. To quantify how

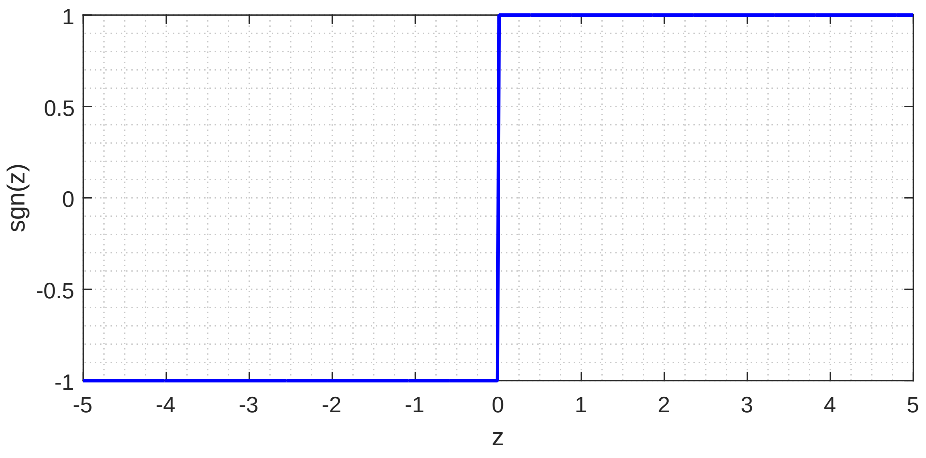


Figure 2.4.: Shape of the step function.

well our weights and biases reach this goal we define the following cost function:

$$C(w,b) = \frac{1}{2n}\sum_x \|y(x) - a(x,w,b)\|^2 \tag{2.5}$$

where w denotes the collection of all weights in the network, b all the biases, n is the total number of training inputs and $a(x,w,b)$ is the vector of outputs from the network which has weights w, biases b and the input x.

So for training our network and finding the best weights and biases we now simply have to solve the problem

$$\arg\min_{w,b} C(w,b). \tag{2.6}$$

This is usually done by applying *gradient descent*. Therefore it is necessary to compute the gradient in every iteration of the gradient descent algorithm. The gradient is of the form $\nabla C = \frac{1}{n}\sum_x \nabla C_x$ where $C_x = \frac{1}{2}\|y(x) - a(x,w,b)\|^2$ for all training inputs x. As it is very time-consuming to calculate the gradient for all the training inputs, one normally uses *stochastic gradient descent* that estimates the gradient by averaging only over a small sample of randomly chosen training inputs in every iteration. The number of randomly chosen samples in every iteration is called *batch size*.

The gradient itself is computed by the so called *backpropagation algorithm* which will not be explained here. A detailed explanation can be found in chapter 2 in [19].

After computing the gradient (for a random batch) and updating the weights for several iterations, one normally computes the *accuracy* of the network with a second set of images for which we also know the desired output. We therefore check how many of these so called *testing images* are already classified correctly by our network. Of course all the testing images must be different from the training images to get a meaningful result.
When we reach an accuracy that is high enough (for example 99%), we can stop training our network.

2.1.3. Preventing Overfitting: Weight Decay & Dropout

In large neural networks we often encounter the problem of *overfitting* if our training set is not large enough. It means that the value of the cost function 2.5 further decreases while the accuracy does not improve anymore or even gets worse. A detailed explanation for the reasons of this can be found in the section "Overfitting and regularization" of chapter 3 in [19].
Overfitting is a major problem in neural networks. This is especially true in modern networks, which often have very large numbers of weights and biases. Increasing the amount of training images is one way of reducing it, but training data are often expensive. Another approach is to reduce the size of the network. However, large networks have the potential to be more powerful than small networks, and so this is an option we'd only adopt reluctantly.
Luckily there are other possibilities to prevent overfitting.

Weight Decay

One possibility is to add a regularization term to our cost function 2.5:

$$C(w,b) = \frac{1}{2n}\sum_x \|y(x) - a(x,w,b)\|^2 + \frac{\lambda}{2n}\sum_w w^2 \tag{2.7}$$

where $\lambda > 0$ is known as the *regularization parameter*. So this new term tries to keep our weights small when we are minimizing our cost function. It is really not at all obvious why this helps to reduce overfitting. An explanation can be found in the section "Why does regularization help reduce overfitting?" of chapter 3 in [19].

Dropout

Dropout is a radically different technique. Unlike regularization, dropout doesn't rely on modifying the cost function. Instead, in dropout we modify the network itself. In particular, with dropout we start by randomly (and temporarily) deleting half the hidden neurons in the network, while leaving the input and output neurons untouched. We forward-propagate the input through the modified network, and then backpropagate the result, also through the modified network. After doing this over a batch of examples, we update the appropriate weights and biases. We then repeat the process, first restoring the dropout neurons, then choosing a new random subset of hidden neurons to delete, estimating the gradient for a different batch, and updating the weights and biases in the network. By repeating this process over and over, our network will learn a set of weights and biases. Of course, those weights and biases will have been learnt under conditions in which half the hidden neurons were dropped out. When we actually run the full network that means that twice as many hidden neurons will be active. To compensate for that, we halve the weights outgoing from the hidden neurons.
For the reason why this also works to prevent overfitting, we want to point again to chapter 3 of [19] where an explanation can be found in the section "Other techniques for regularization".

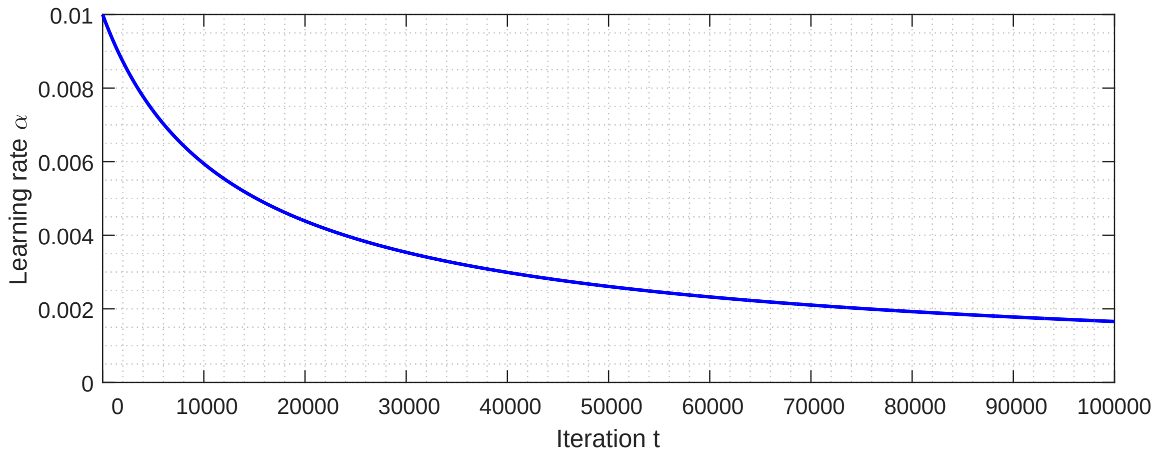


Figure 2.5.: Plot of our learning rate for 100,000 iterations if we use $\alpha_0 = 10^{-2}$, $\beta = 0.75$ and $\gamma = 10^{-4}$.

2.1.4. Hyper-Parameters

We have seen how neural networks look like and how they work. One of the biggest arts when using neural networks for some specific task is to choose a number of neurons and a number of hidden layers which result in the behaviour that one wants to have. Also choosing the step size for the gradient descent (usually referred to as *learning rate* in neural networks) and a batch size that perform good for learning is mostly not trivial.
It is also possible to use a learning rate which changes during training. It makes sense to decrease the learning rate with an increasing number of iterations, because a small step size is better if we are close to a minimum of our optimization problem whereas we need a bigger step size if we are far away from a minimum.
We are useing the following rule for the learning rate α:

$$\alpha = \alpha_0 \cdot (1 + \gamma \cdot t)^{-\beta} \tag{2.8}$$

where the base learning rate α_0, γ and the power β are hyper-parameters and t is the current number of iteration. A plot of this function can be found in Figure 2.5. The weight update V_{t+1} in iteration $t+1$ is then computed by the following formula:

$$V_{t+1} = \mu \cdot V_t - \alpha \nabla L(W_t) \tag{2.9}$$

where the *momentum* μ is another hyper-parameter and W_t are the weights and biases in iteration t. The new weights and biases W_{t+1} can now be computed by

$$W_{t+1} = W_t + V_{t+1}. \tag{2.10}$$

We will always use base learning rate $\alpha_0 = 10^{-2}$, power $\beta = 0.75$, $\gamma = 10^{-4}$, momentum $\mu = 0.9$ and regularization parameter $\lambda = 5 \cdot 10^{-4}$.

2.2. Convolutional Neural Networks

In the last section, we have described neural networks that consist exclusively of fully connected layers. Fully connected means that every neuron in one layer is connected to every neuron in the previous layer and to every neuron in the next layer. We have seen this for example in Figure 2.1. Convolutional networks do not only consist of fully connected layers, but also of *convolutional layers* that can extract features that are *local* and *equivariant* much more efficiently. Local means that the features only depend on some neighbouring input neurons, equivariant means roughly speaking that it does not matter where in the input the features appear. This is often very useful when processing images which we represent as 2 dimensional maps of the input pixels as already mentioned in the previous chapter.

We are now going to describe convolutional layers shortly. For a more detailed explanation, take a look at chapter 11 in [21].

2.2.1. Convolutional Stage

As already mentioned, in convolutional layers the neurons are not fully connected, but only *sparsely connected*. The number of input neurons that every neuron in one convolutional layer has is called *kernel-size*. In Figure 2.6 we have a sparse connectivity with kernel-size 3 on the left hand side and in contrast a full connectivity on the right hand side.

If we have a 2 dimensional input like a (greyscale) picture, then the kernel is also 2 dimensional and we would have a kernel-size of 5×5 for example.
The output is calculated by a *convolution operation* that will not be discussed in detail here.

Another property of convolutional layers is the fact that the kernel has tied weights. This means that the kernel shares the weights and bias at every position. In Figure 2.7 we can see how weight sharing works.

As we can already see in Figure 2.7, the convolutional operation is shrinking the network

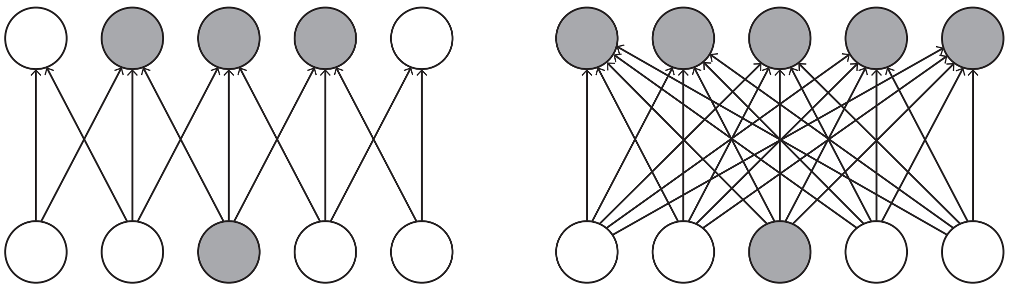


Figure 2.6.: Here we can see sparse connectivity (left) vs. full connectivity (right).

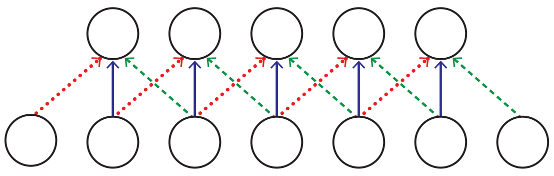


Figure 2.7.: Here we can see an illustration of weight sharing. All connections that have the same color, also share the same weights. The red (dotted), the blue and the green (dashed) connections share weights.

by the size of the kernel minus 1. In this case we have a kernel of size 3, so the network is shrinking by the size of 2. To avoid this shrinking, we can use *zero-padding* which makes the network artificially wider by adding implicit zeros. In Figure 2.8 we can see an example of zero-padding.

In the example of Figure 2.8 we have just added as many zeros as necessary to keep the network the same size (this is called *same convolution*). But of course this results in input neurons near the boarder influencing fewer output neurons than input neurons near the center. Because of that sometimes even more zeros are added in order to underrepresent the border pixels less.

Another parameter of convolutional layers is the *stride*. When the kernel is moved from one position to the next, the step-size is called stride. So normally we have a stride of 1, but it is for example also possible to choose a stride of 2. Then every second output neuron is skipped which also means that the size of the network reduces to half of the input size.

As it is only possible to detect one feature if we have one kernel that shares weights, one normally applies several different kernels and thus gets different outputs for different features. The number of different kernels that we apply is called *filter-size*. This means that a filter-size greater than 1 increases the dimension of the network.
So for example if we use zero padding to keep the network the same size and also use a stride of 1 to not reduce the size of the network, but have a filter-size of 4, we get 4 feature

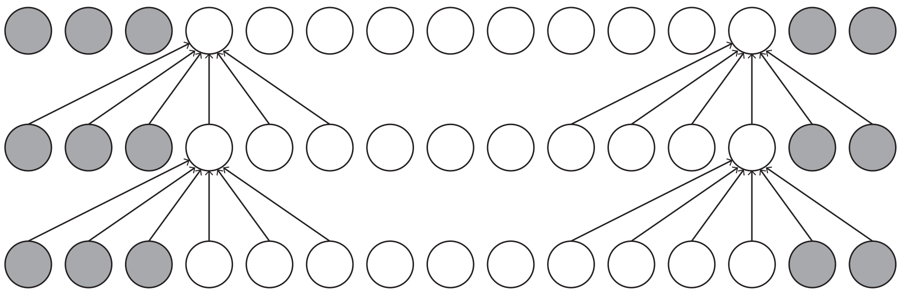


Figure 2.8.: Same convoluiton as an example of zero padding for a kernel-size of 6. [21]

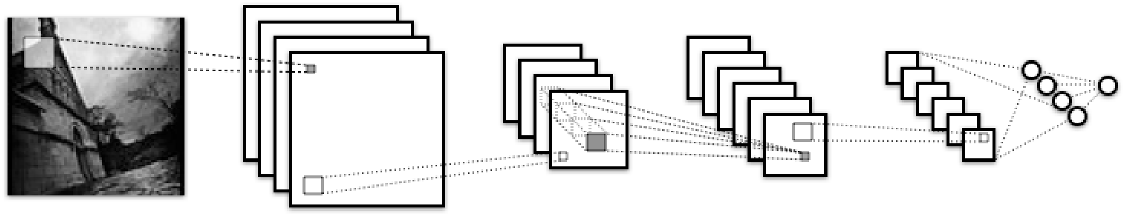


Figure 2.9.: Here we can see a visualization of the feature maps of a Convolutional Neural Network. In the first layer we have 4 feature maps that have the same size as the original image. In the second layer we have 4 smaller feature maps, in the second layer 6 even smaller feature maps and in the third layer again 6 feature maps that are again even smaller. At the end we have two fully connected layers. [22]

maps that all have the same size as the input. See Figure 2.9 for an example.

Normally a convolutional layer not only performs the convolution stage that we have just described, but often additionally consists of a *detector stage*, a *pooling stage*, or both. So in literature there are two different terminologies: Sometimes the three stages are summarized as one convolutional layer and sometimes they are represented as three different layers. In Figure 2.10 we have a comparison of the two different representations.

In the following we are going to describe the detector and pooling stage shortly.

2.2.2. Detector Stage

In the detector stage, every input is run through a nonlinear activation function, such as for example the rectified linear activation function:

$$f(x) = \max(0, x) \tag{2.11}$$

where x is the input to a neuron and $f(x)$ the output. This operation supports in-place computation, meaning that the input and the output layer could be the same to preserve memory consumption, as the output at every position only depends on the input at the same position.

2.2.3. Pooling Stage

Pooling layers help to make the output invariant to small translations of the input. This is very useful if we are more interested if a certain feature is present than where exactly it is.

A pooling function takes some neighbouring inputs and summarizes them to one output. So for example the *max pooling* operation on a 2 dimensional input (for example an image), takes a rectangular of inputs and outputs the maximum value. The size of the rectangular is called *kernel-size* just like in the convolutional stage that we have described

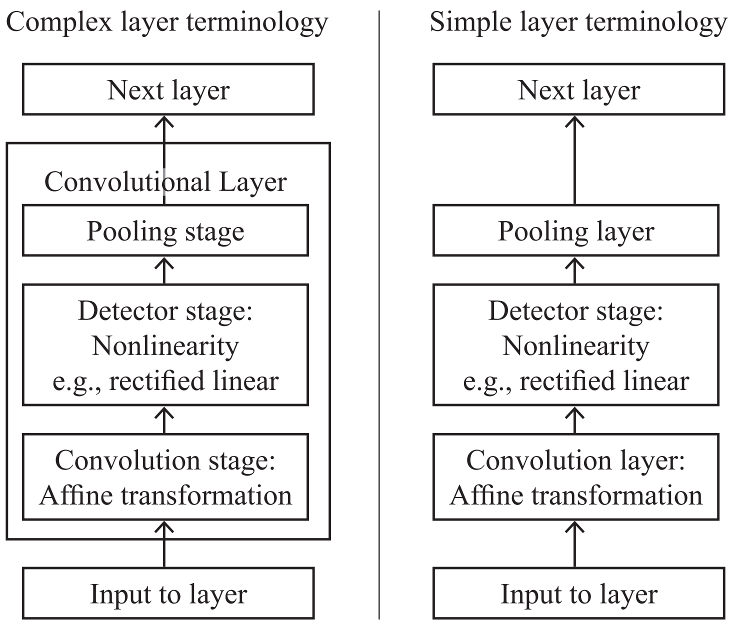

Complex layer terminology
Next layer
Convolutional Layer
Pooling stage
Detector stage: Nonlinearity e.g., rectified linear
Convolution stage: Affine transformation
Input to layer
Simple layer terminology
Next layer
Pooling layer
Detector stage: Nonlinearity e.g., rectified linear
Convolution layer: Affine transformation
Input to layer

Figure 2.10.: There are two terminologies to describe the stages of a convolutional layer. One of them summarizes the three stages as one layer (left) and the other represents them as three different layers (right).

above. Also the parameters *zero-padding* and *stride* are exactly the same for pooling layers. Other examples of pooling functions are the average of a rectangular neighborhood, a weighted average based on the distance from the central pixel, or the L2 norm of a rectangular neighborhood.
Because pooling summarizes the output over a whole neighborhood of inputs, often a stride grater than 1 is used in order to reduce the spatial size of the representation. This improves the computational efficiency of the network, because the next layer has fewer inputs to process.

Part II.

CAPTCHA Recognition with Deep Neural Networks

3. A Deep CNN for CAPTCHA Recognition

We propose a deep CNN to solve the whole sequence of a `CAPTCHA`. Our main purpose is to combine the segmentation and detection stage in this network. We first state the architecture of our neural network in Section 3.1 and then we explain how we classify multi-digit characters with this one network in Section 3.2.

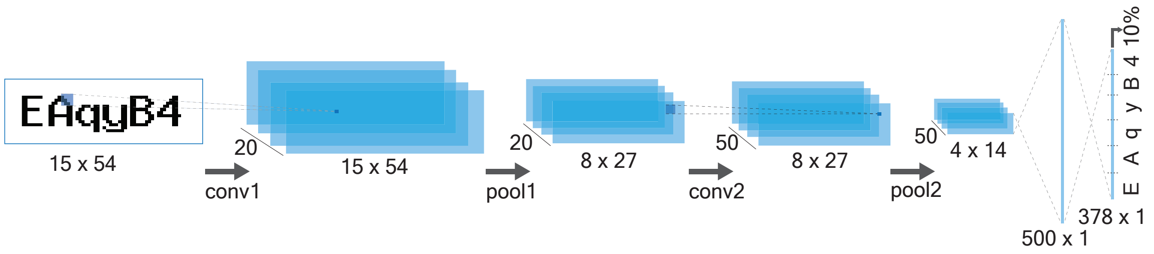


Figure 3.1.: **Convolutional Neural Network for `CAPTCHA` Recognition.** Our CNN is composed of two convolution, two pooling and two fully-connected layers. The last layer outputs the length and the probability distributions for all digits.

3.1. Network Architecture

As shown in Figure 3.1 our network is composed of two convolutional and pooling layers, one fully connected layer (which is called inner product in Caffe) and one output layer that is the *classifier*. The last layer is designed to output the length of the `CAPTCHA` and probability distributions for each digit. The network is apart from the input and output size identical to the LeNet MNIST Tutorial from Caffe[1]. The first convolutional layer has a size of 20, the second one a size of 50. They both have a kernel size of 5×5. The first fully connected layer has a size of 500 and the second fully connected layer (classifier) has an output size of 378. The pooling layers have a window size of 2×2 and a stride of 2. The first fully connected layer contains rectified linear units. The batch size for every iteration is 64. The layout of this network is illustrated in Figure 3.2.

We use this network for all of our tests with the easy `CAPTCHA`s, but later on when we solve more challenging ones, we will also add more layers and increase their size.

[1] `http://caffe.berkeleyvision.org/gathered/examples/mnist.html`

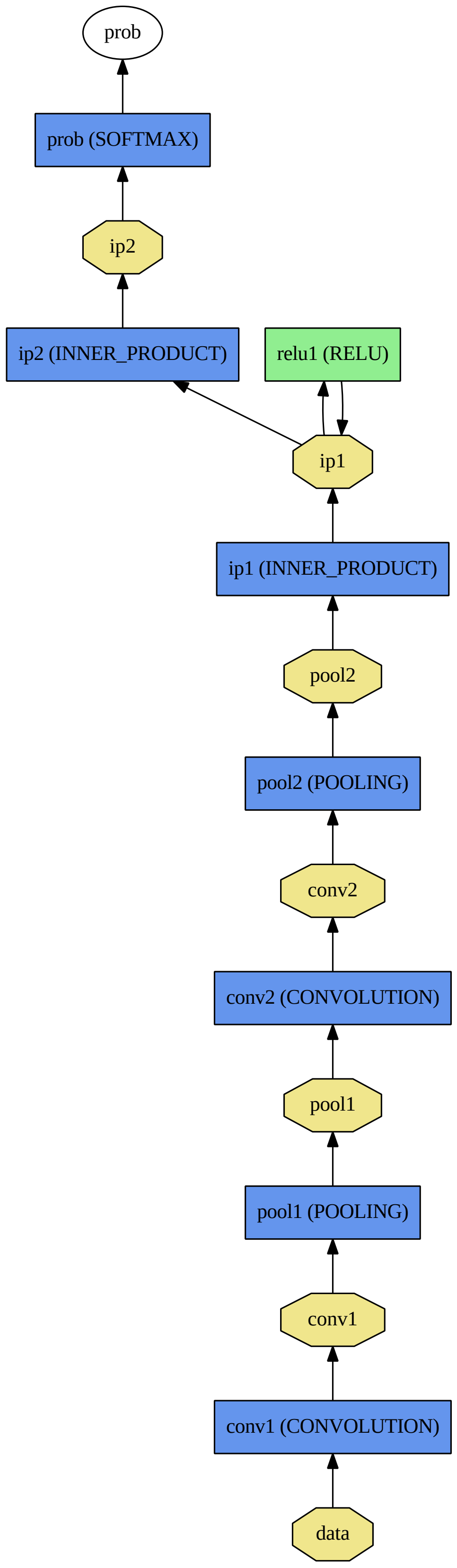


Figure 3.2.: Visualization of our network for the easy `CAPTCHA`s.

3.2. Multi-digit Character Recognition

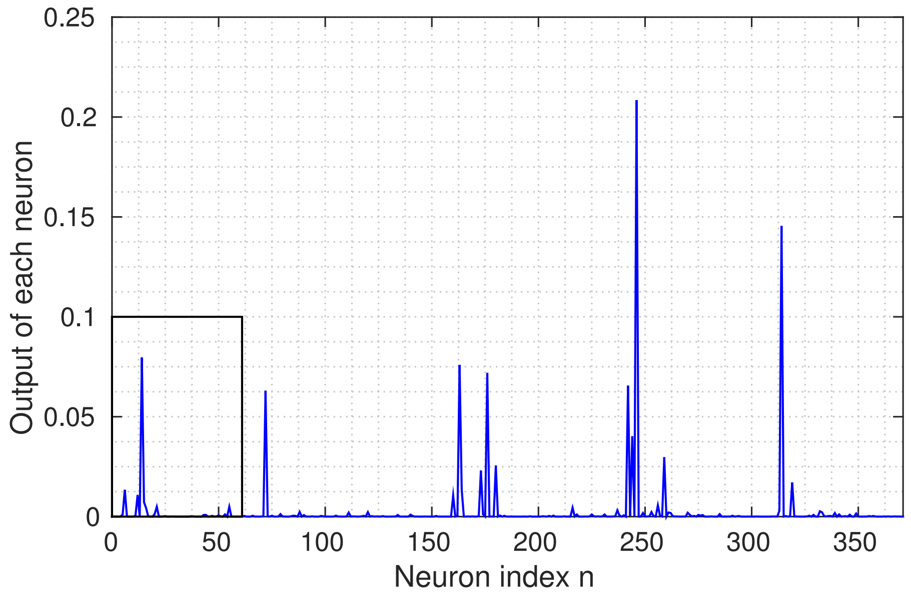
In order to classify multiple digits in our CAPTCHAs, we need to define a maximum number of digits. We choose a maximum of 6. Each of these digits is then represented by 62 neurons in the output layer of our network. We define a bijective function $\Theta(x)$ that maps every digit ($x \in \{$'0',...'9', 'A',..., 'Z', 'a',..., 'z'$\}$) to a unique integer $l \in \{0, ...61\}$:

$$\Theta(x) = \begin{cases} 0 \ldots 9, & \text{if x = '0' \ldots '9'} \\ 10 \ldots 35, & \text{if x = 'A' \ldots 'Z'} \\ 36 \ldots 61, & \text{if x = 'a' \ldots 'z'} \end{cases} . \tag{3.1}$$

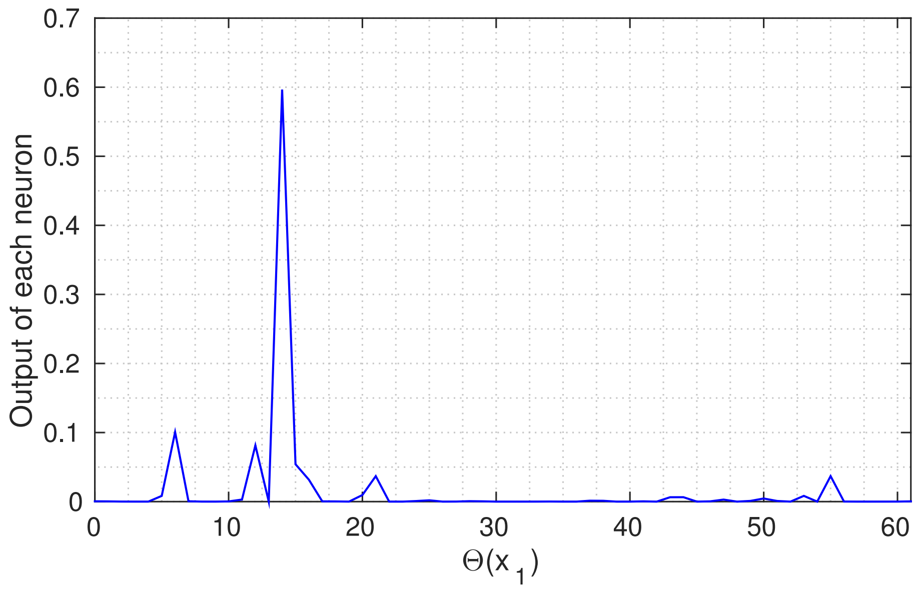
We assign the first 62 neurons of the output layer to the first digit of the sequence, the second 62 output neurons to the second digit and so on. To predict a digit, we only look at the corresponding 62 neurons and normalize the sum to 1 to estimate the probability distribution. For example, as shown in Figure 3.3, if the predicted character index for the first digit is $c_0 = 14$, then the predicted label is $x_1 = \Theta^{-1}(c_0) =$ 'E'. If the predicted character index for the second digit is $c_0 = 62 + 10 = 73$, then the predicted label is $x_2 = \Theta^{-1}(c_0 - 1 \cdot 62) =$ 'E'.
The corresponding digit (x_i) for each neuron index (n) is computed as follows:

$$x_i = \Theta^{-1}(c_0 - i \cdot 62) \tag{3.2}$$

where $i \in \{0, ..., 5\}$ is the index of the digit. The output layer has $6 \cdot 62 = 372$ neurons for the digits. Since we also recognize the length of the CAPTCHA, we add one more output neuron for every possible length. So in our case, we have in total 378 neurons in the output layer if we allow CAPTCHAs of length 1 to 6.
If we now want to classify an image, we take the output of our network and first of all take a look at the last six classes. The class with the highest output value is the most probable class and we assume that this is the length of the sequence. In our example if the output of class 377 would be higher than the output of the classes 373, 374, 375, 376 and 378, then we assume that the length of the sequence is 5. In this case we would ignore the sixth 62 classes that correspond to the sixth character.



(a)



(b)

Figure 3.3.: **Network output.** (a) Output of the network for the exemplar CAPTCHA "EAqyB4" from Figure 3.1. There are 62 outputs for each digit. The black box shows the output for the first digit. (b) Probability distribution for the first digit with the sum normalized to 1.

4. Reducing Training Data by Active Learning

Training a deep CNN usually requires a very large hand-labeled dataset to obtain a high classification accuracy. However, in reality it is infeasible to collect millions of hand-labeled samples for `CAPTCHA` recognition. Therefore, we propose to use an Active Learning framework (see Figure 4.1). The main idea of this approach is to add new training data only if it is necessary, i.e. if the sample is informative enough for re-learning. This is decided based on the uncertainty of the prediction, and we show different ways to compute uncertainty in Section 4.1. Then, in Section 4.2 we describe the differences of our Active Learning application over other methods, in particular the way we incorporate ground truth information more efficiently.

The difference between online learning and active learning is the following. Online learning simply refers to algorithms that can learn sequentially without having to re-consider the whole training set. So, ideally this means we observe a new sample (or many) and directly use them for learning without re-learning on the old samples. This is useful if data becomes available in a sequential fashion and one does not have access to the entire training dataset at once from the beginning. In contrast, active learning is able to decide what samples to learn on. This normally means that the learning algorithms can actively

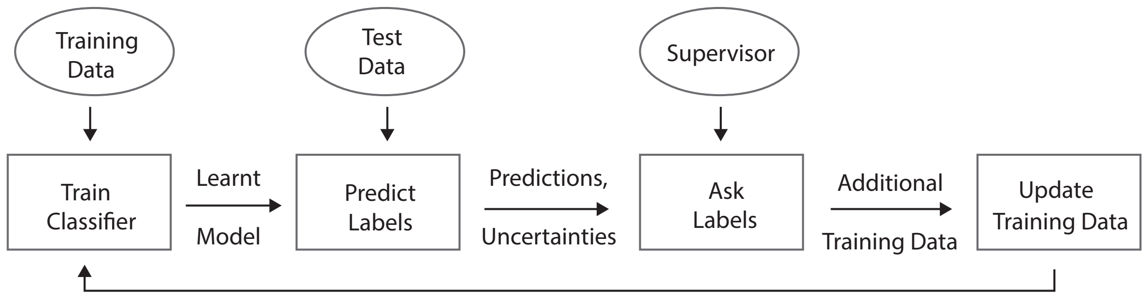


Figure 4.1.: Flow chart of Active Learning. We start with training on a small data set. Then, the classifier is applied to some new data, and the result is a label prediction and an associated uncertainty. Based on this uncertainty, the classifier decides whether a ground truth label should be asked for the new sample or not. Note, in our case this query is simply performed by solving a given `CAPTCHA` and using it for learning if the uncertainty is too high. In that case, the training data is increased and learning is performed again. In our method, we use a deep CNN, which can be efficiently re-trained using only the newly added training samples.

query a human for the correct label of some data that he thinks it is very important to learn from. If the algorithm chooses the examples in a clever way, the overall number of examples that are necessary to achieve a certain accuracy can often be much lower than the number required in normal supervised learning.
We are using a combination of both, active and online learning.

4.1. Obtaining Uncertainty

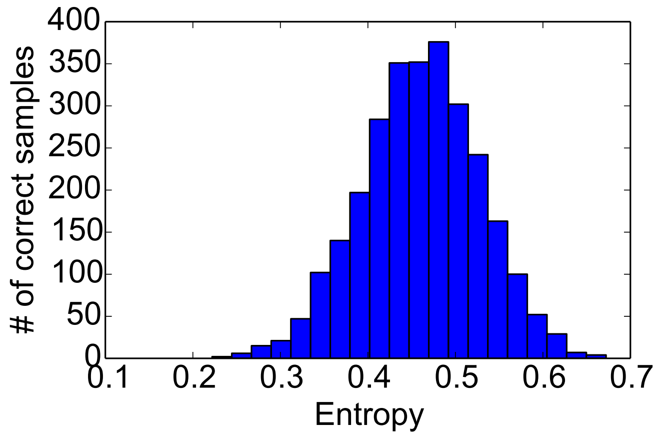
To evaluate the quality of the prediction, we compute the uncertainty of the prediction from the output layer. The network estimates the probability distribution for each digit as explained in Section 3. For each digit we only consider the corresponding neurons and normalize the sum of all the outputs to 1 as shown in Figure 3.3. We consider 3 different methods to compute the uncertainty η:

- Normalized entropy:
$$\eta = -\frac{1}{d} \cdot \sum_{i=1}^{d} \sum_{j=1}^{k} p_j \cdot \log_k(p_j). \tag{4.1}$$
where d is the total number of digits in a sequence, k is the number of all possible cases for one digit (62 in our settings) and p_j is the predicted probability for the case j. The entropy is 0 if the prediction is certain and 1 otherwise.

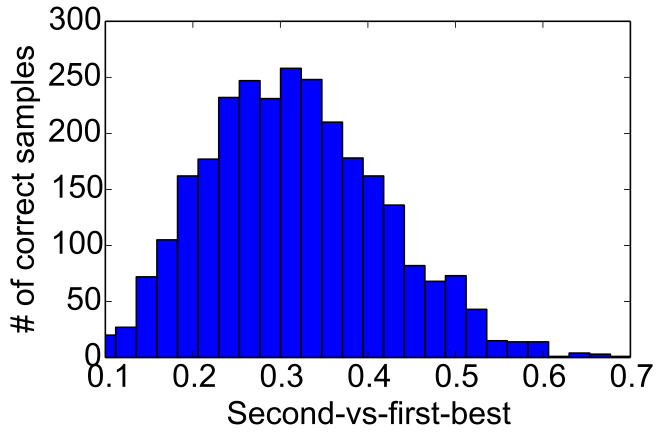
- Second-vs-first-best:
Let $\mathbb{P}_{x_i}$ be the probability distribution for the digit x_i. Then the uncertainty of the sequence is computed as follows:
$$\eta = \frac{1}{d} \cdot \sum_{i=1}^{d} \frac{\arg\max \{\mathbb{P}_{x_i} \setminus \arg\max \mathbb{P}_{x_i}\}}{\arg\max \mathbb{P}_{x_i}} \tag{4.2}$$
where the numerator is the second best and the denominator is the best prediction for the digit x_i.

- Squared mean error as uncertainty:
$$\eta = \sum_{i=1}^{d} \frac{(1 - \arg\max \mathbb{P}_{x_i})^2}{d}. \tag{4.3}$$

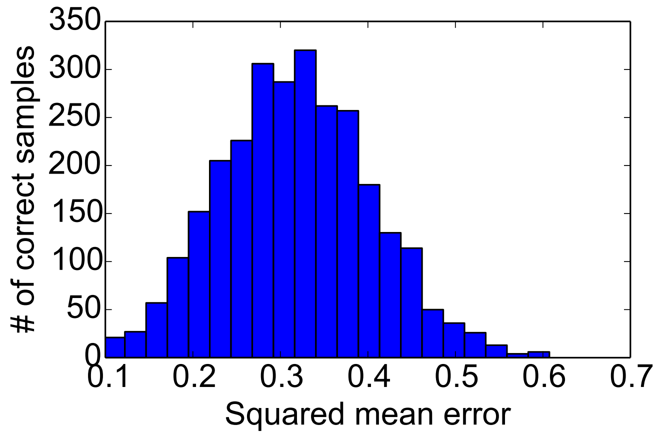
Figure 4.2 shows the uncertainty histograms for the correctly classified images from 10,000 test samples with a network trained on 2,500 CAPTCHAs for 10,000 iterations with dropout, where we used all three different methods to compute uncertainty. Intuitively, the classifier should associate correct classifications with only very little uncertainty. Therefore, we choose to use the second-vs-first-best strategy, as it yields the smallest average uncertainty for the correct labels. Note that in principle, we also have to consider the uncertainty histograms of the incorrect classified samples, because for a new test sample we do not have a ground truth yet, and we do not know whether the classifier is correct or not. However,



(a)



(b)



(c)

Figure 4.2.: **Uncertainty histograms.** The histograms are computed for the correctly classified images from 10,000 test samples for the three methods: (a) normalized entropy, (b) second-vs-first-best and (c) squared mean error as uncertainty.

in our special case we can easily obtain this information by trying to solve the `CAPTCHA`, as described in the next section. Thus, only the correct predictions are interesting, because for them we do have a ground truth label.

The *average uncertainty* of one testing set for a network is then the mean uncertainty of all correctly classified images in the testing set.

4.2. Obtaining Ground Truth Information

As mentioned above, our `CAPTCHA` recognition problem is unique in the sense that we can perform learning without human intervention. We achieve this by only using those data samples for re-training, for which the classifier already provided a correct label. For these, the `CAPTCHA` can be solved and we know what the correct text is. However, simply using all these correctly classified samples for re-training would be very inefficient. In fact, training would be done more and more often, because the classifier will be better over time and therefore classify more samples correctly. Thus, with every new correctly classified sample a retraining would be necessary. To avoid this we use the uncertainty values presented above: only if the prediction uncertainty is higher than a threshold, we use that sample for re-training. This has the effect, that over time the classifier will reduce its uncertainty and therefore require less data for re-training.

Of course in theory it would make much more sense to not learn from the correctly classified `CAPTCHA`s, but from the wrong ones. But as already mentioned, in practice this is not possible, because we do not get the labels for the wrong ones. If we try to solve a `CAPTCHA` on some webpage, we only get correct or wrong as a result.

Part III.

Experimental Results

5. Dataset Generation

As there is no hand-labeled CAPTCHA dataset, *e.g.* for Google's reCAPTCHA which is the most widely used CAPTCHA provider in the world [23], we generate two different CAPTCHA datasets.

5.1. Easy CAPTCHAs

First of all we want to test our method with easy CAPTCHAs. Therefore we create sequences with undistorted text and a maximum length of 6. We have written an own script that creates the images that all have a dimension of 54×15. For simplicity we fix the length to 6 for the first experiments, but we also show how to solve variable length ones later in Section 6.4. Some examples of our easy auto-generated CAPTCHAs are illustrated in Figure 5.1.

5.2. Cool PHP CAPTCHAs

After having a look at our easy CAPTCHAs, we also want to try out more challenging ones. So we are going to use the "Cool PHP CAPTCHA" framework [24] to generate test and training images. These CAPTCHAs are composed of more challenging distorted text similar to Google's reCAPTCHA. We have modified the framework to not generate colored pictures, but black and white ones. Furthermore we have disabled shadows and the line through the text. We also do not use dictionary words, but random words. Therefore we have modified the class and removed the rule that every second character has to be a vowel, but we keep the original setting to only use small letters, but not 'k's and 'u's. Otherwise it is not even possible for a human to distinguish for example a small 'v' from a big 'V'. We fix the font to "AntykwaBold". For simplicity we additionally fix the length of the sequence to 6. The images always have an dimension of 180×50. Figure 5.2 shows some auto-generated examples.

During the auto-generation, we ensure that there is no duplication in the datasets.

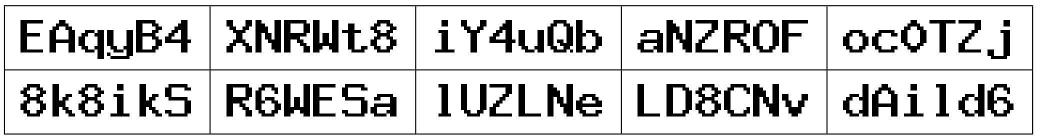


Figure 5.1.: Some examples of our Easy CAPTCHAs that we use in the experiments. They are composed of undistorted text.

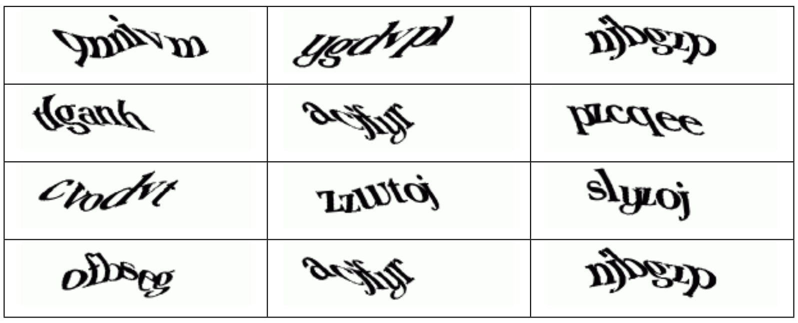


Figure 5.2.: Some examples of the Cool PHP CAPTCHAs that we use in the experiments. They consist of distorted text.

6. Evaluation

In this chapter we present our results on auto-generated CAPTCHAs as described in chapter 5. All experiments have been executed using the Caffe [18] deep learning framework on an NVIDIA GeForce® GTC™ 750 Ti GPU. To our best knowledge, it is the most appropriate framework available to train deep neural networks and also open source. On their own website they state that "Speed makes Caffe perfect for industry use. (...) We believe that Caffe is the fastest CNN implementation available." [25]

We first of all solve the easy CAPTCHAs in Section 6.1, then we apply Active Learning in Section 6.2. Afterwards, we show our results for the more challenging CAPTCHAs in Section 6.3 and last but not least, we solve variable length ones in Section 6.4.

6.1. Recognition of the easy CAPTCHAs

For the easy CAPTCHAs, we create the network as described in Section 3.1.

First of all we train it with 50,000 training images for 10,000 iterations. Afterwards we test the network with 10,000 testing images (which are all different from the training images as already mentioned). The training takes less than 3 minutes on our hardware and the accuracy that we achieve is 98.68%. For all the remaining 1,32% wrong classified CAPTCHAs, there was only one wrong classified digit while all the other five ones were classified correctly.
After training for 10,000 further iterations (which takes again less than 3 minutes), we achieve an accuracy of 100%.

If we add padding of size 2 to each side of our two convolutional layers, we already get an accuracy of 99.83% after 10,000 iterations. We explain the reason for that later in Section 6.2.2.

6.2. Reducing the Initial Training Set

As the most expensive part is to get the training samples, we aim at decreasing the required size of the training set in this section.

First of all, we train our network with a decreased size of the training set with only 2,500 training images. After every 1,000 iterations, we test the accuracy with 10,000 testing images. After 3,000 iterations we achieve an accuracy of 9.83% which even decreases with more iterations. In Figure 6.1 there is a plot of the accuracy for 20,000 iterations.

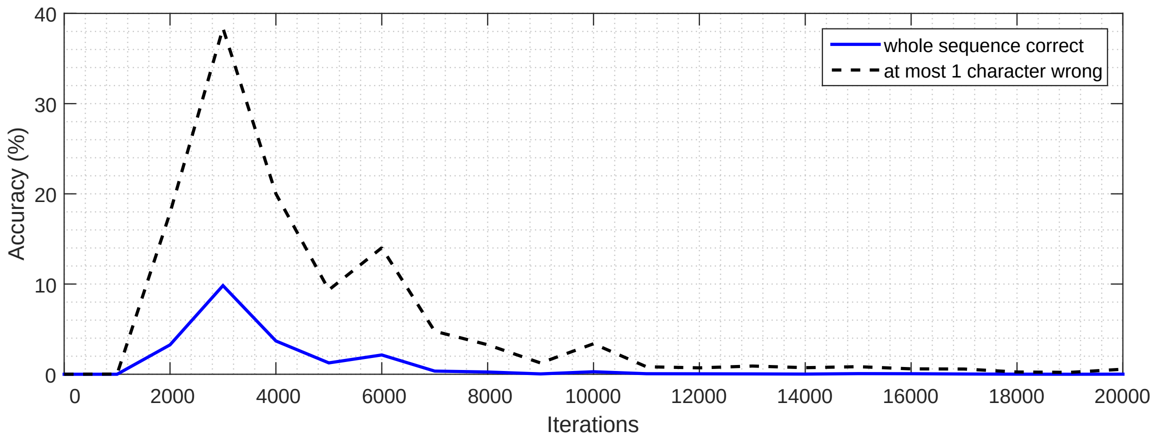


Figure 6.1.: Accuracy plot for 20,000 iterations if we train our network without dropout with only 2,500 training images.

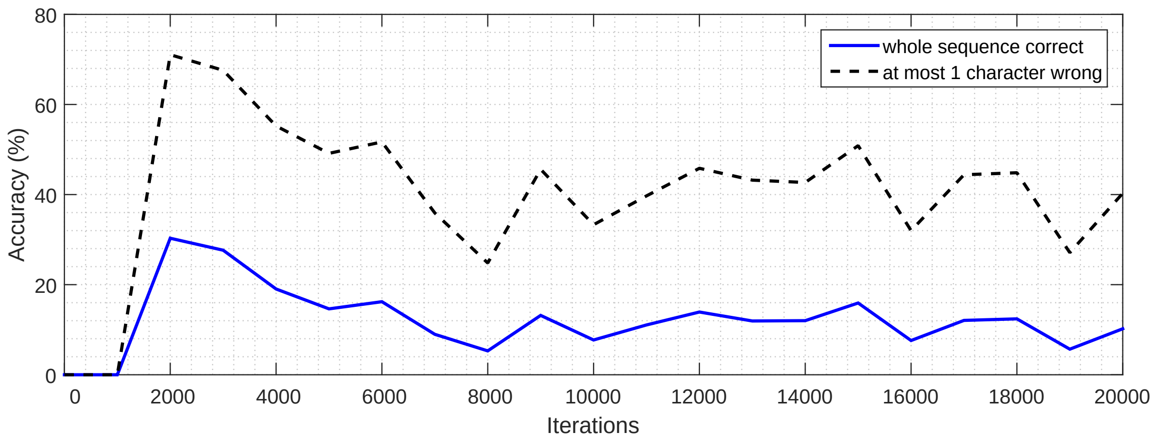


Figure 6.2.: Accuracy plot for 20,000 iterations if we add dropout to our network and train with only 2,500 training images.

As we can see, the network is obviously overfitting. So in the next step, we add dropout (50%) to both of the convolutional layers and the first fully connected layer of our network (we do not add dropout to the second fully connected layer which is our classifier).
The result for 20,000 iterations is plotted in Figure 6.2.

We can see that the result is slightly better now, but still very bad. Because of that, we now want to make use of Active Learning as described in chapter 4.

6.2.1. Increasing the Training Set with Active Learning

So first of all, we train our network with 5,000 training images for 10,000 iterations with the same network as previously without dropout. Afterwards we want fo find new training samples by trying to solve 10,000 CAPTCHAs. Then we add all of the correctly classified

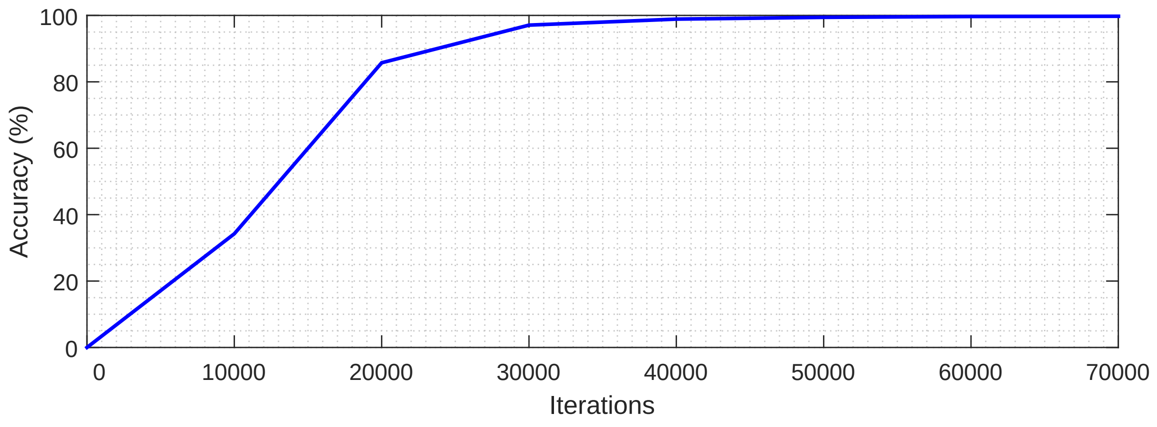


Figure 6.3.: Accuracy plot if we start to train our network with only 5,000 training images, but try to classify 10,000 images after every 10,000 iterations and add the correctly classified ones to the training set.

ones to the training set and retrain again for 10,000 iterations. We iterate over this method for 100,000 iterations. When we test the accuracy after every 10,000 iterations with 10,000 testing images, we get the graph as displayed in Figure 6.3.

As we can see, we get much better results than before. After the first 10,000 iterations, we again have a quite low accuracy of $\sim 30\%$. But after adding around 3,000 images to our training set, we already achieve an accuracy of 86% after 10,000 additional iterations. The accuracy further increases to more than 99% after the next iterations.

After 30,000 iterations we have an accuracy of 97.08% and the confusion matrices for all the six characters are displayed in Figure 6.4 for 10,000 testing images.
A confusion matrix shows the quality of the classification of one single character. Each column of the matrix represents the instances in a predicted class, while each row represents the instances in an actual class. So for every image in the testing set, we check for this character which class is predicted by our network (and which class is the correct one) and then increase this position in the matrix by 1. We additionally normalize the matrix by dividing every column by the sum of this column. Then every entry in the matrix is a value between 0 and 1. If the network predicts everything perfectly correct, we then get the identity matrix. The name "Confusion Matrix" comes from the fact that it makes it easy to see if the system is confusing two classes (i.e. commonly mislabeling one as another).

We see that the classification only has problems with the first character while all the others are classified perfectly correct. We see that the first character is for example often misclassified as 'b' (class 37) instead of 'p' (class 51) or the other way round, or an 'h' (class 43) is misclassified as an 'n' (class 49) or the other way round, or an 'o' (class 50) is misclassified as a 'p' (class 51) or the other way round.
In Figure 6.5 we have `CAPTCHA`s on top of each other that start exactly with the characters that we have just mentioned. What we realize is that those characters are identical every-

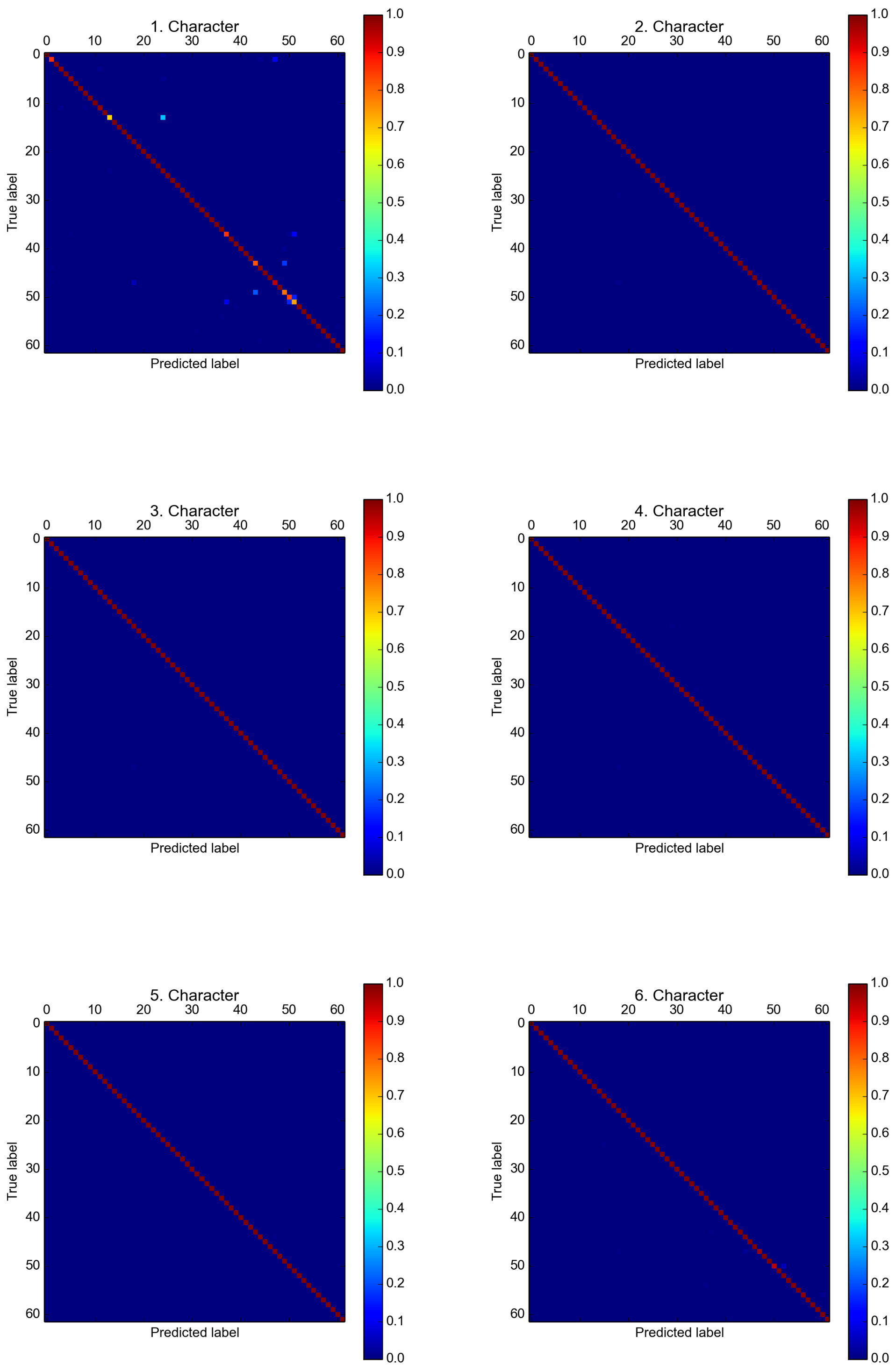


Figure 6.4.: Confusion matrices for the 6 characters of our 10,000 training images after 30,000 iterations if we add new images to the training set after every 10,000 iterations.

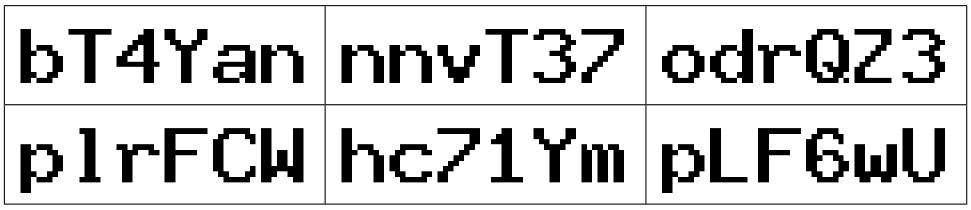


Figure 6.5.: If we compare the first character of two CAPTCHAs on top of each other, we realize for our easy CAPTCHAs that apart from the very first row of pixels, a 'b' is identical to a 'p', an 'h' is identical to an 'n' and an 'o' is identical to a 'p'.

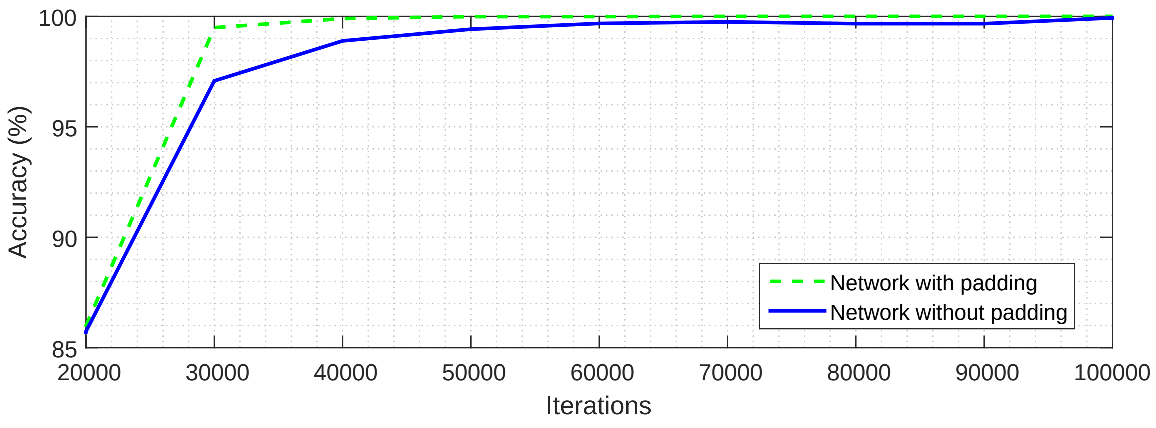


Figure 6.6.: Here the blue line is identical to the one from Figure 6.3 (starting from 20,000 iterations). It indicates the accuracy if we add the correctly classified CAPTCHAs to the test set after every 10,000 iterations.
The green (dashed) line indicates the accuracy if we do exactly the same, but add padding of size 2 to our network.

where apart from the very first row of pixels.

So it is assumable that the first row of pixels is underrepresented and we add padding of size 2 to our network in a next step. in Figure 6.6 there is a plot that illustrates the improvement. Now we already get an accuracy of 99.49% after 30,000 iterations. So our assumption was correct that the outer pixels were underrepresented and padding solves this issue.

This is also the explanation why we get better results with padding in Section 6.1.

6.2.2. Using Uncertainty to choose the Training Data with Active Learning

In order to further improve the performance, we want to make use of Active Learning without increasing the size of the training set in every iteration. We train our network with only 2,500 training images for 10,000 iterations with dropout. Afterwards, we try to classify 50,000 testing images, but only pick 5,000 of the correctly classified ones for our

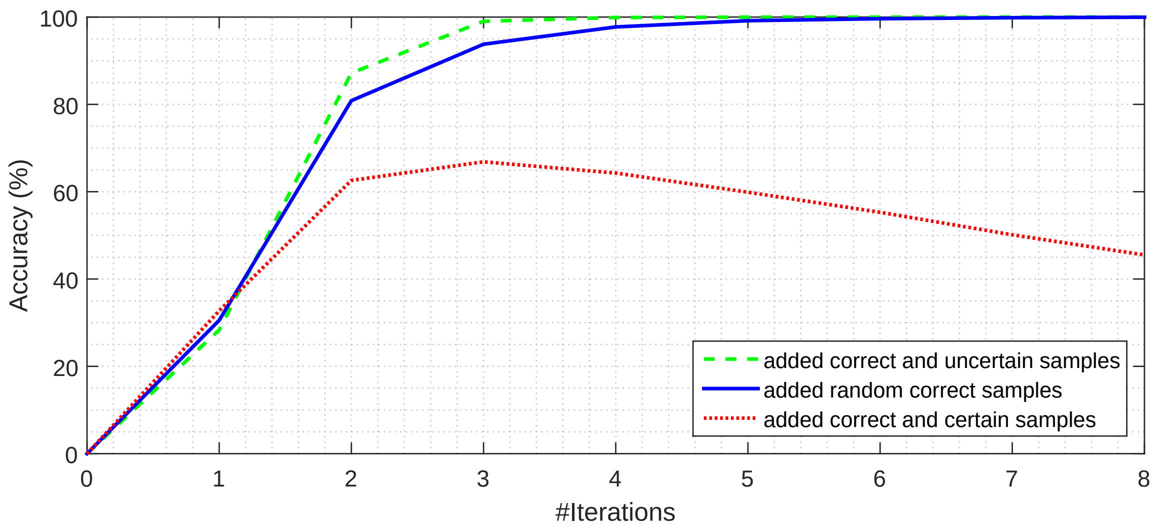


Figure 6.7.: **Improvement of the accuracy with Active Deep Learning.** We choose 5,000 new samples out of 50,000 images at every iteration to fine-tune the network. Each curve shows the accuracy for the corresponding chosen set. Using the correct and uncertain samples gives the best performance with less required training data.

next training iteration without including the images from the previous iteration. We can pick the samples with 1) the highest uncertainty, 2) the lowest uncertainty, or 3) randomly. For computing the uncertainty, we take our second-vs-first-best uncertainty as described in Section 4.1. We then train once again with dropout for 10,000 iterations. Subsequently we do the same procedure to get a new training set of 5,000 images. Afterwards, we continue to train without dropout and only for 1,000 iterations with every new training set. We apply this algorithm for in total 8 iterations. The accuracy is computed after each iteration on a fixed test set that consists of 10,000 images. We run the algorithm 10 times for each of the data selection strategies and plot the average of all the 10 tests in Figure 6.7.

If we always add the correct images with the lowest uncertainty, we get the worst result (red, dotted curve in Figure 6.7). This is because we show the images to the network that it already classified correctly and also has a high certainty on them. We see that the network is not able to learn from them anymore after 3 iterations and the accuracy even decreases afterwards. In case that we add 5,000 random images from the correct classified ones, we have a satisfying result (blue curve in Figure 6.7). The accuracy already converges to 100%. However, the number of training samples to achieve 100% accuracy is higher than if we choose to take the 5,000 correctly classified images with the highest uncertainty (green, dashed curve in Figure 6.7). We receive 100% accuracy with relatively less training data. This is reasonable as the network in fact classifies the images correctly, but still is very uncertain about the classification. Hence it can learn from the fact that it was indeed right with its classification. Obviously the learning effect should be even better if we could learn from the wrong classified images. This is indeed the case, but not possible in practice.

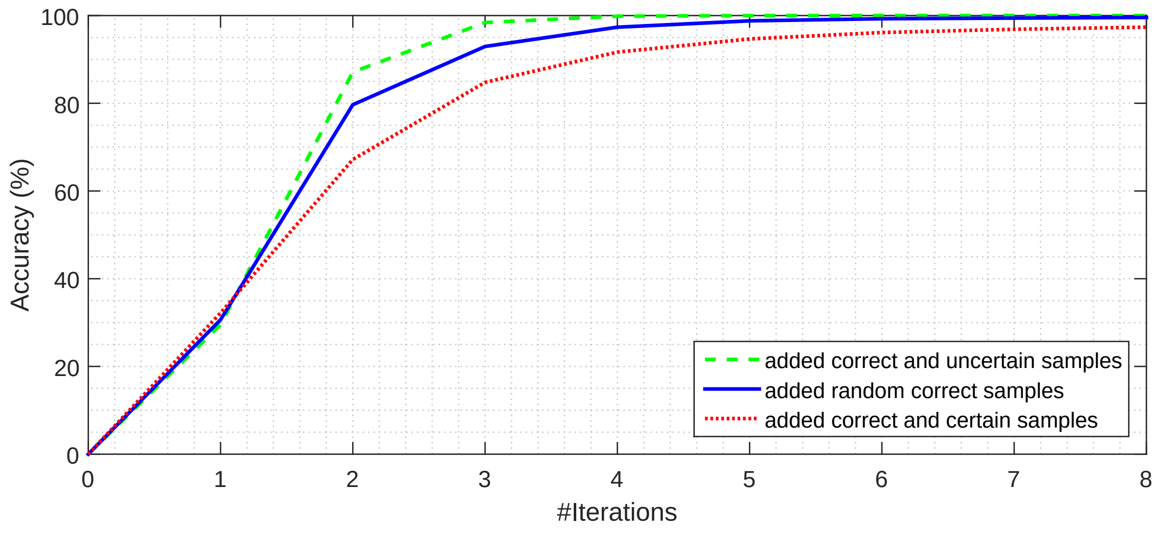


Figure 6.8.: **Choosing new training samples with thresholding.** New training set of 5000 images in every iteration is built with thresholding. Using the correct and uncertain samples gives the best performance again.

Another way of using our Active Learning method is to define a threshold. Therefore we again train our network with dropout and 2,500 initial training images for 10,000 iterations. Afterwards, we try to classify `CAPTCHA`s one after each other. If a classification is correct and the uncertainty is higher than the threshold, we add the image to the training set for the next iteration. We repeat this until we have a new training set of size 5,000 and then train once again with dropout for 10,000 iterations. Subsequently we do the same to get 5,000 new training images in every iteration and train without dropout and only for 1,000 iterations with every new training set. For the most uncertain classified ones, we choose an initial threshold of 0.35 and decrease it by 50% for every new training set. We take a fixed threshold of 0.2 for the most certain ones and not any threshold if we pick randomly, because we can just take the first 5,000 correctly classified images. The results are shown in Figure 6.8. We again plot the average out of 10 runs.
Using a threshold to choose new samples allows the network to learn even with correct and certain ones. However, choosing a smaller threshold would yield worse results similar to our previous experiment (compare Figure 6.7).

6.3. Recognition of the Cool PHP CAPTCHAs

We now want to try to solve the more challenging Cool PHP CAPTCHAs. In order to be as flexible as possible, we do not apply any cropping. The Cool PHP CAPTCHA project also provides the possibility to put a line through the text and put grey background strips on the image. We do not use these features, but if we wanted to solve those even more complicated CAPTCHAs in a next step, then cropping would not be possible anymore as we do not know where the text is.

We want to start with a network with 3 convolutional and pooling layers. The convolutional layers have a size of 48, 64 and 128. Then the network has one fully connected layer of size 3072 and a second fully connected layer (classifier) that has an output size of 372 again. The convolutional layers have a kernel size of 5×5, we again add a padding of 2 and the first convolutional layer additionally has a stride of 2. The pooling layers have a window size of 2×2 and the first and the third one also have a stride of 2. The first fully connected layer contains rectified linear units. The batch size for every iteration is 64.

When we test the accuracy of a network in the following, we always test with 50,000 images after every 10,000 iterations.

First of all we train the network as described above with 100,000 training images for 100,000 iterations. This takes around 8 hours on our hardware. The result that we get for this network is plotted in Figure 6.9. The blue curve indicates the actual accuracy (that the whole sequence was predicted correctly) whereas the black (dashed) curve indicates the accuracy if we allow one character to be wrong if all the five other characters are classified correctly. We achieve an accuracy of 49.03% after 50,000 iterations, but it drops down to 20.27% after 100,00 iterations. So here we are obviously overfitting.

To prevent overfitting, we first of all try to increase the training set and train the same

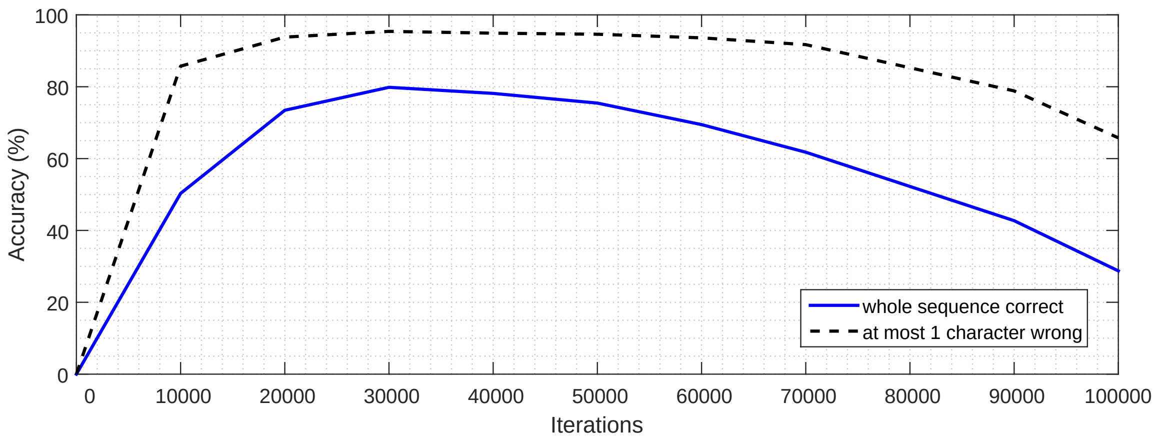


Figure 6.9.: Accuracy if we train our network with 3 convolutional layers with 100,000 training images.

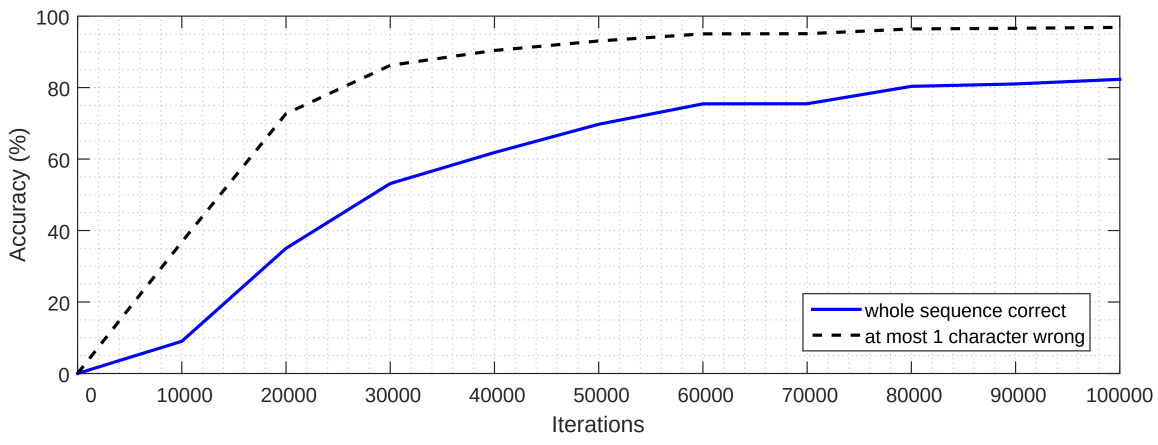


Figure 6.10.: Accuracy if we train our network with 3 convolutional layers with 500,000 training images.

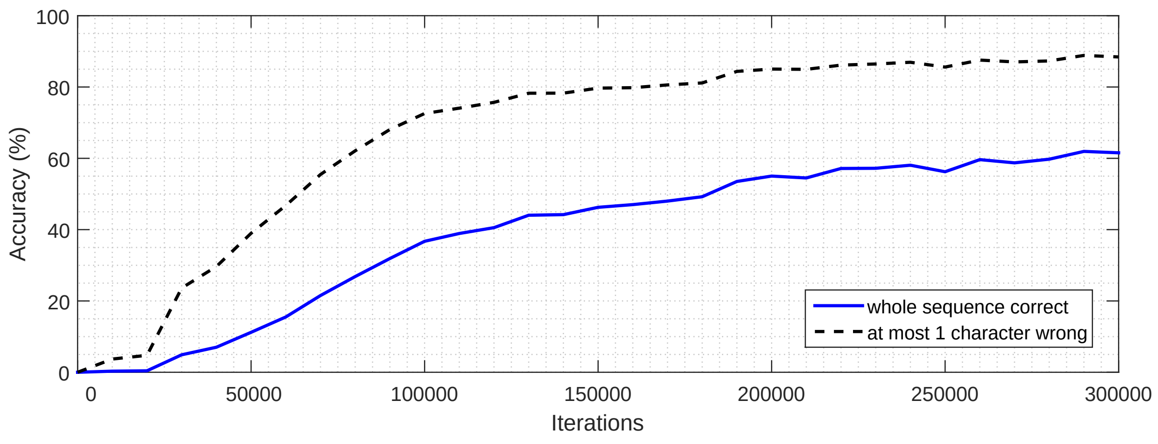


Figure 6.11.: Accuracy if we train our network with 3 convolutional layers and dropout with 100,000 training images.

network with 500,000 training images. The accuracy that we get is plotted in Figure 6.10. We are not overfitting anymore and we achieve an accuracy of 82.31% after 100,000 iterations (96.85% that at most one character is wrong).

But as we cannot increase the training set in any order if we increase our network, we want to try to prevent overfitting by adding dropout. So we train our network again with only 100,000 training images, but we add dropout (50%) to both of the convolutional layers and the first fully connected layer of our network. The result is plotted in Figure 6.11. We are not overfitting anymore, but we only achieve an accuracy of 61.55% (88.43% that at most one character is not classified correctly) after 300,000 iterations.

In a next step, we additionally add rectified linear units after every layer. With 100,000 training images again, we then achieve an accuracy of 80.75% (95.54 that at most one char-

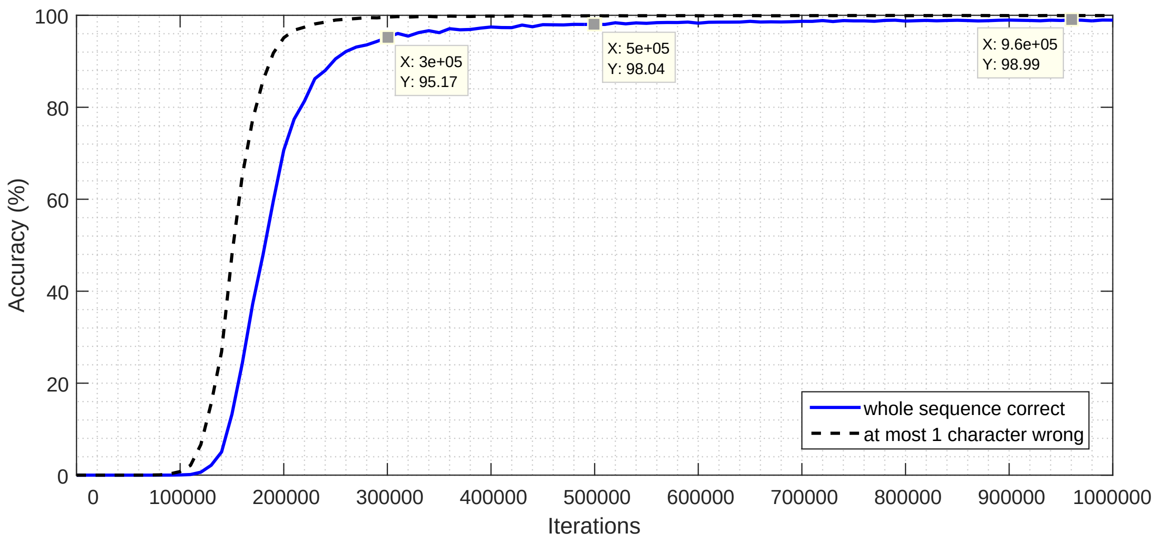


Figure 6.12.: Accuracy if we train our network with 7 convolutional layers with 500,000 training images.

acter is classified wrong). So from now on we will always use dropout and rectified linear units for all layers expect for the last one which is our classifier.

In the next step, we add two additional convolutional and pooling layers after the third one and also increase the size of the other ones a little bit. In our new network the convolutional layers have size 60, 80, 150, 200 and 250. The first, third and fifth pooling layers and also the first convolutional layer have a stride of 2. All the other settings remain the same. When we train this network with 500,000 images for 300,000 iterations, we achieve an accuracy of 97.28% that the whole sequence is correct and an accuracy of 99.84% that at most one character is wrong. This takes approximately 46 hours on our hardware.

Last but not least, we add even two more convolutional and pooling layers to our network. The pooling layers both have a size of 250 and neither the pooling nor the convolutional layers have a stride grater than 1. The accuracy that we achieve with one million iterations of training is displayed in Figure 6.12. It first stays 0% for nearly 100,000 iterations. After 300,000 iterations, which already take around 2.5 days of training, the accuracy achieves 95.17%. With 500,000 iterations it already increases to 98.04%. At last with nearly 1,000,000 iterations which take approximately 8 days of training, the accuracy reaches 98.99%. Here we have an accuracy of 99.95% that at most one character is wrong.

6.4. Recognition of Variable Length CAPTCHAs

For recognizing variable length CAPTCHAs, we want to use our easy CAPTCHAs again. We create 100,000 training images and 50,000 testing images with a random length of 4, 5, or 6. We train the same network as described in Section 6.1 with two convolutional layers and two fully connected layers. We use the network with padding of 2 and without any dropout. We do the classification of variable length sequences as described in Section 3.

If we training our network for 10,000 iterations (which takes approximately 5 minutes on our hardware), we achieve an accuracy of 99.97%. All the images that our network classifies wrong have a length of 6 and the error is in the last sixth character. The reason for this is that we train the sixth character the fewest. We train the first four characters for all CAPTCHAs in the training set, the fifth character only for sequences of length 5 or 6 and the sixth character only for CAPTCHAs of length 6.
After 20,000 iterations of training (which takes around 10 minutes), we achieve an accuracy of 100% for all of our 50,000 testing images.
The accuracy includes predicting the correct length of the sequence as well as predicting all characters correctly.

Part IV.

Conclusion and Outlook

7. Conclusion

We propose a CAPTCHA solving technique that is able to solve easy CAPTCHAs as well as hard ones with distorted text. Furthermore we apply an Active Learning algorithm that uses initially a very small set of images to train a deep Convolutional Neural Network and then improves the classifier by exploiting the test samples. New training samples are chosen from the test set based on their uncertainty. Our quantitative results show that the performance of the network can be significantly improved with the correctly classified but uncertain test samples.

7.1. Outlook

In the meantime lots of CAPTCHAs have been updated to not show distorted text, but some other task. Some examples can be found in Figures 7.1, 7.2, 7.3 and 7.4.

Also Google updated their CAPTCHAs to the so called "No CAPTCHA reCAPTCHA" system. With a single click of the user on the CAPTCHA, the system tries to detect if the click comes from a robot. This is done with the help of the mouse movements when the user moves the mouse to select that box, the IP address and the browser cookies. In case that the system is not sure if the click comes from a human, a normal CAPTCHA is shown which has also been improved especially for mobile devices. So for example images are shown that have to be classified, as demonstrated in Figure 7.5. But this task can also be perfectly solved with deep CNNs (compare LeNet [10]) and can thus be perfectly improved with our Active Learning method.

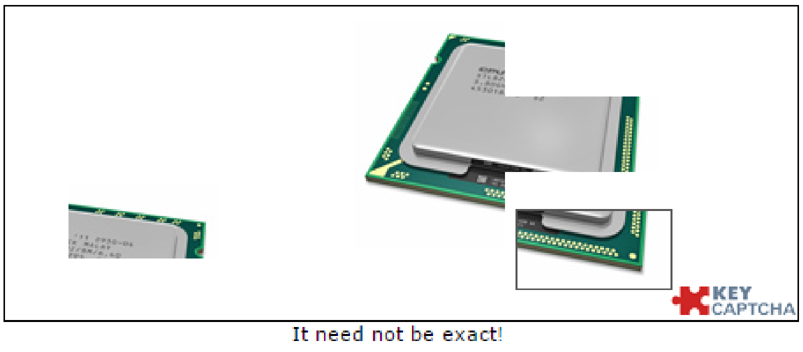


Figure 7.1.: In this CAPTCHA the user has to move and put together the image parts with the mouse like in a jigsaw puzzle.

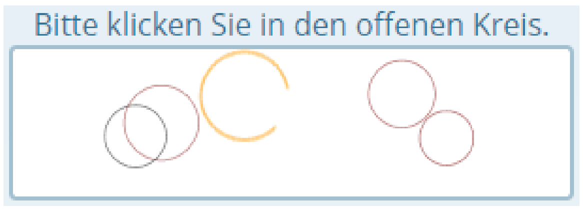


Figure 7.2.: In this CAPTCHA the user has to click inside the open circle.

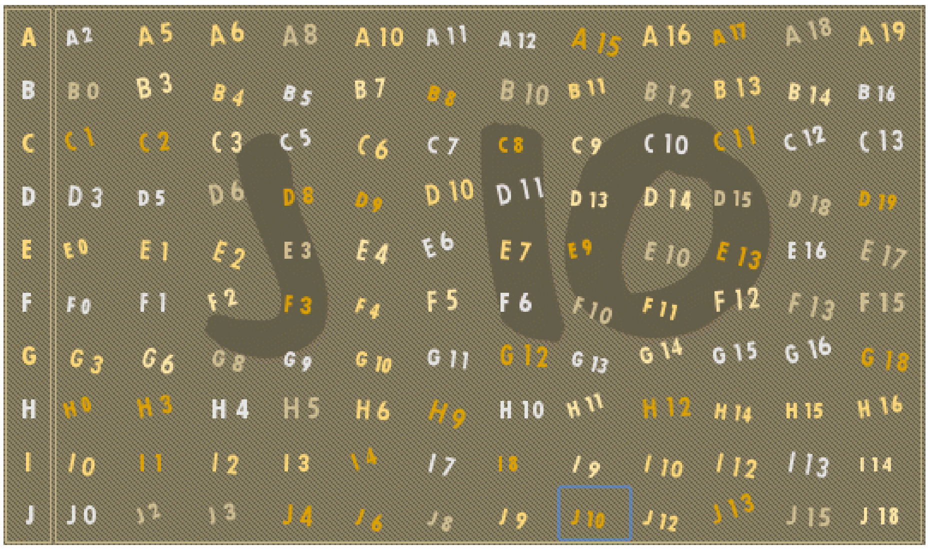

J10
A A2 A5 A6 A8 A10 A11 A12 A15 A16 A17 A18 A19
B B0 B3 B4 B5 B7 B8 B10 B11 B12 B13 B14 B16
C C1 C2 C3 C5 C6 C7 C8 C9 C10 C11 C12 C13
D D3 D5 D6 D8 D9 D10 D11 D13 D14 D15 D18 D19
E E0 E1 E2 E3 E4 E6 E7 E9 E10 E13 E16 E17
F F0 F1 F2 F3 F4 F5 F6 F10 F11 F12 F13 F15
G G3 G6 G8 G9 G10 G11 G12 G13 G14 G15 G16 G18
H H0 H3 H4 H5 H6 H9 H10 H11 H12 H14 H15 H16
I I0 I1 I2 I3 I4 I7 I8 I9 I10 I12 I13 I14
J J0 J2 J3 J4 J6 J8 J9 J10 J12 J13 J15 J18

Figure 7.3.: In this CAPTCHA the user has to read the background ("J10" in this special case) and then click on that text in the foreground (the blue box in this example).

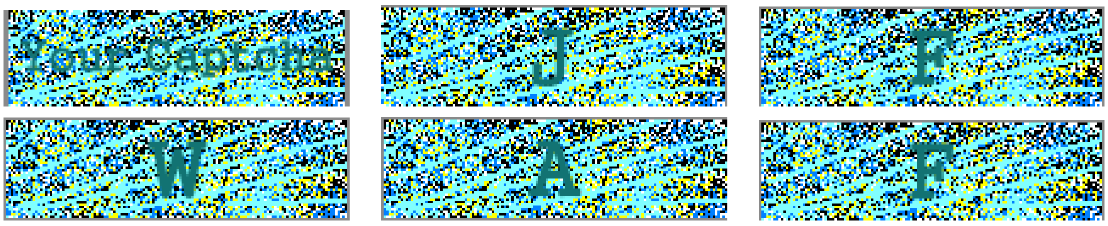


Figure 7.4.: This CAPTCHA is animated and shows all of these images for one second one after each other. The user has to type all of the letters afterwards. In this example: "JFWAF".

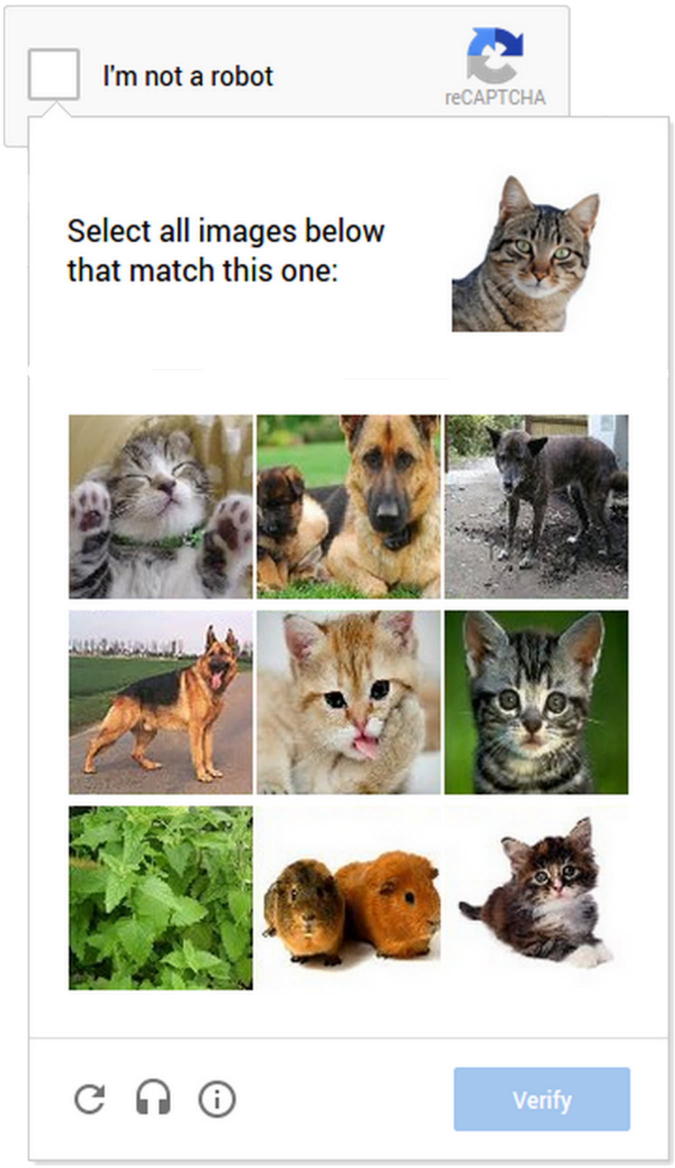


Figure 7.5.: One example of Google's new `No CAPTCHA reCAPTCHA` [26]. Here the user has to select all images that show a cat.

Bibliography

[1] Luis von Ahn, Manuel Blum, Nicholas J Hopper, and John Langford. Captcha: Using hard ai problems for security. In *EUROCRYPT*, 2003. 3, 4

[2] PY Simard. Using machine learning to break visual human interaction proofs (hips). *Advances in neural information processing systems*, 17:265–272, 2005. 3, 4

[3] Ian J Goodfellow, Yaroslav Bulatov, Julian Ibarz, Sacha Arnoud, and Vinay Shet. Multi-digit number recognition from street view imagery using deep convolutional neural networks. *arXiv preprint arXiv:1312.6082*, 2013. 3, 4, 5

[4] Dan Claudiu Ciresan, Ueli Meier, Luca Maria Gambardella, and Juergen Schmidhuber. Deep big simple neural nets excel on handwritten digit recognition. *arXiv preprint arXiv:1003.0358*, 2010. 3

[5] Adam Coates, Andrew Y Ng, and Honglak Lee. An analysis of single-layer networks in unsupervised feature learning. In *International Conference on Artificial Intelligence and Statistics*, pages 215–223, 2011. 3

[6] George E Dahl, Dong Yu, Li Deng, and Alex Acero. Context-dependent pre-trained deep neural networks for large-vocabulary speech recognition. *Audio, Speech, and Language Processing, IEEE Transactions on*, 20(1):30–42, 2012. 3

[7] Geoffrey Hinton, Li Deng, Dong Yu, George E Dahl, Abdel-rahman Mohamed, Navdeep Jaitly, Andrew Senior, Vincent Vanhoucke, Patrick Nguyen, Tara N Sainath, et al. Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *Signal Processing Magazine, IEEE*, 29(6):82–97, 2012. 3

[8] Yoshua Bengio, Réjean Ducharme, Pascal Vincent, and Christian Janvin. A neural probabilistic language model. *The Journal of Machine Learning Research*, 3:1137–1155, 2003. 3

[9] Ronan Collobert and Jason Weston. A unified architecture for natural language processing: Deep neural networks with multitask learning. In *Proceedings of the 25th international conference on Machine learning*, pages 160–167. ACM, 2008. 3

[10] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In *NIPS*. 2012. 3, 45

[11] Max Jaderberg, Andrea Vedaldi, and Andrew Zisserman. Deep features for text spotting. In *Computer Vision–ECCV 2014*, pages 512–528. Springer, 2014. 4

[12] Yi Lu. Machine printed character segmentation-; an overview. *Pattern Recognition*, 28(1):67–80, 1995. 4

[13] Patrice Y Simard, Dave Steinkraus, and John C Platt. Best practices for convolutional neural networks applied to visual document analysis. In *2013 12th International Conference on Document Analysis and Recognition*, volume 2, pages 958–958. IEEE Computer Society, 2003. 4

[14] Greg Mori and Jitendra Malik. Recognizing objects in adversarial clutter: Breaking a visual captcha. In *Computer Vision and Pattern Recognition, 2003. Proceedings. 2003 IEEE Computer Society Conference on*, volume 1, pages I–134. IEEE, 2003. 4

[15] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 1998. 4

[16] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Andrew Senior, Paul Tucker, Ke Yang, Quoc V Le, et al. Large scale distributed deep networks. In *Advances in Neural Information Processing Systems*, pages 1223–1231, 2012. 4

[17] Guangjie Shi, Yuchen Ying, and Yue Yin. The recaptcha helper: A machine learning study. 5

[18] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093*, 2014. 5, 31

[19] Michael A. Nielsen. Neural networks and deep learning. *Determination Press*, 2015. 7, 8, 10, 11

[20] Alexey Dosovitskiy, Jost Tobias Springenberg, Martin Riedmiller, and Thomas Brox. Discriminative unsupervised feature learning with convolutional neural networks. In *Advances in Neural Information Processing Systems*, pages 766–774, 2014. 7

[21] Yoshua Bengio, Ian J. Goodfellow, and Aaron Courville. Deep learning. Book in preparation for MIT Press, 2014. 13, 14

[22] Convolutional neural networks (lenet). `http://deeplearning.net/tutorial/lenet.html`. Accessed: 2015-04-20. 15

[23] Google recaptcha. `https://www.google.com/recaptcha`. Accessed: 2015-04-17. 29

[24] cool-php-captcha. `https://code.google.com/p/cool-php-captcha/`. Accessed: 2015-04-17. 29

[25] Caffe. `http://caffe.berkeleyvision.org`. Accessed: 2015-04-17. 31

[26] Google onlince security blog - are you a robot? introducing "no captcha recaptcha". `http://googleonlinesecurity.blogspot.de/2014/12/are-you-robot-introducing-no-captcha.html`. Accessed: 2015-05-22. 47