MDig: Multi-digit Recognition using Convolutional Nerual Network on Mobile

Xuan Yang
Stanford University
Stanford, CA 94305
xuany@stanford.edu

Jing Pu
Stanford University
Stanford, CA 94305
jingpu@stanford.edu

Abstract

Multi-character recognition in arbitrary photographs on mobile platform is difficult, in terms of both accuracy and real-time performance. In this paper, we focus on the domain of hand-written multi-digit recognition. Convolutional neural network (CNN) is the state-of-the-art solution for object recognition, and presents a workload that is both compute and data intensive. To reduce the workload, we train a shallow CNN offline, achieving 99.07% top-1 accuracy. And we utilize preprocessing and segmentation to reduce input image size fed into CNN. For CNN implementation on the mobile platform, we adopt and modify DeepBeliefSDK to support batching fully-connected layers. On NVIDIA SHIELD tablet, the application processes a frame and extracts 32 digits in approximately 60ms, and batching the fully-connected layers reduces the CNN runtime by another 12%.

1 Introduction

With the large number of the hand-written documents, there is a great demand to convert the hand-written documents into digital record copies, which are more accessible through digital systems, such as digital forms and databases. To automatically accomplish the transformation from hand-written numbers to their digital version, multi-digit recognition is inevitable and useful.

Convolutional neural network (CNN) is a state-of-the-art image recognition technique in the community. Prior researches have shown single digit recognition can achieve less than 1% error rate [1, 2, 3, 4] using CNN. For Multi-digit recognition, there are studies about recognizing house numbers, vehicle VIN numbers and so on [5, 6]. However, to our best knowledge, multi-digit recognition using CNN on mobile device has not been well studies.

A mobile solution has many advantages for this problem, as it is portable, cheap, and has friendly interface for scanning the documents. However, besides the general requirement for high accuracy and robustness over environment changes, the mobile platform has its own constraints, such as real-time responses, limited memory resources. In particular, running CNN on mobile has been a challenging problem, since a traditional CNN often requires a humongous mount of memory.

Although having a powerful server at the back-end to process and recognize the images can greatly reduce the computation of the client, the solution suffers from an extra network latency and requires a reliable network connection. Therefore, it will still be preferred to have a client-based solution. In order to achieve the performance requirements on the mobile client, we optimize our system in the following ways:

- **Segmenting digit patches**. We reduce the computation in the recognition step by preprocessing and segmenting the digit from the original images, and only feed the digit patches to the CNN for recognition.
- **Simple CNN architecture**. Compared with the state-of-the-art deep CNNs [7, 8, 9] which are designed for complex object recognition, we build a shallow network with fewer kernels and train it just for hand-written digit recognition. The simple CNN requires much less memory and runs fast on mobile, and yet our result shows that it can still achieve similar accuracy – less 1% error rate.
- **Batching fully-connected layers**. Batching the computation the fully-connected layer for several images trades off the computation efficiency against the single image latency. The latter is not important for our multi-digit recognition task, where we aim at the low latency of recognizing all the digits in the frame. Therefore, we implement batched fully-connected layer for our CNN, which has more parameter reuse and better cache locality.

Our results show that with a relatively shallow CNN, we can achieve 99.07% top-1 accuracy for single digit recognition on MNIST dataset [10]. By using the above optimization approaches, we can process a frame and extract 32 digits in approximately 60ms. Batching fully-connect layer reduces the CNN runtime by another 12%.

2 Related work and contribution

2.1 Related work

Convolution neural network has shown it strong capability for several computer vision problems. Many CNN implementations have focused on recognizing a single object in a queried image [9, 8, 11]. Our problem targets at recognizing multiple objects in a single pass, since a number contains multiple digits, and people could write multiple numbers on the same document as well.

Some early work used Spatial Displacement Neural Network (SDNN) combined with Hidden Markov Model to recognize the digits in a string [12]. However, the full string is fed into the neural network, which will still be computationally intensive for mobile devices. In addition, for multiple number strings, the system has to run more than one passes. The recent implementations used Recurrent Neural Network, where the output gets updated based on the recognition result of a latest feed-in slice of the image [13]. This can achieve good performance on a powerful machine, but still subject to the above issues.

Google integrated localization and segmentation steps with deep convolutional neural network to recognize multi-digit numbers in street view and achieved high accuracy [5]. However, this requires the training dataset to have multi-digit in a single image, and suffers from large computation workload. In our approach, we segment the digit patch at first to reduce the computation workload for recognition, and the output of each digit patch is combined at the end.

Batching is well-known optimization to speedup runtime and improve cache locality for neural networks (NN). A recent work has investigated batching NN for speech recognition and the trade-off of the throughput against latency increase [14]. The result shows that NN on CPU achieves comparable performance to GPU, when batching all the layers to convert the linear computation to matrix multiplication. In a CNN, however, the fully-connected layers are only connected at the end of the network, and convolutional layers do not benefit from batching. Instead, batching will increase cache usage and deteriorate cache locality. Therefore, we use steaming in convolutional layer, and batching in fully-connected layer.

2.2 Contribution

In this project, we make the following contributions:

- We build an mobile application that can recognize multiple hand-written digits in a single pass and report the numbers. It works under different light conditions, and different pen drawings.
- We build and train a custom CNN network offline, achieving high validation accuracy. We modified the DeepBeliefSDK to support our CNN architecture on the mobile platform.
- We adapted several optimization techniques to speedup the application, including segmenting digit patches, batching fully-connected layers. We also provide a comprehensive analysis about the application performance.

3 Technical approach

3.1 System pipeline

As introduced in Section 1, mobile application is subject to the limited memory and low latency requirement, we adapt several techniques to overcome those limitations. The multi-digit recognition process consists of:

- **Preprocess** image to grey scale, and use Canny edge detection to locate the digits so that we can only amplify digits and set the background to all black to reduce noise, .
- **Segment** digit patches with using contour finder and rescale it to 28×28, which is used for recognition. Besides, the system also computes which digits belong to the same number based on the position of digits.
- **Recognition** of the digit from each patch using CNN. The CNN is trained off-line, and parameters is preloaded to mobile devices. We batch multiple images in fully-connected layer to speed-up the CNN computation.

3.2 Preprocessing

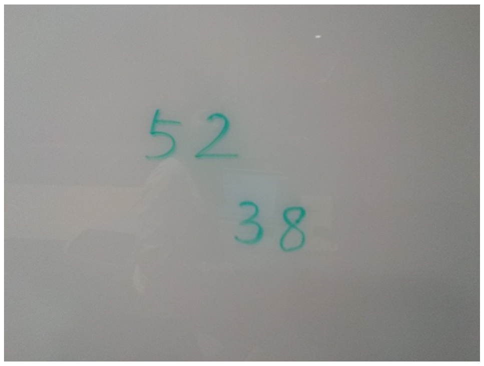
Users are required to take photos of hand-written numbers on light-colored paper or board. However, with the real world lighting, the shadows and specular highlights make it difficult to segment and recognize the digits directly. For example, in Figure 1(a), the intensity values of the digits are close to the shadow, so applying a global threshold to image cannot segment the digits from the background effectively. We address the issue by first running a preprocessing procedure on the taken image. We found the preprocessing step useful since it can eliminate the noise comes with the paper and light and only amplify the digits.

In the preprocessing, Canny edge features computed on the image are fed into contour finder to draw the bounding box of each feature. The result of the bounding box is shown in Figure 1(b). To improve the speed of preprocessing step, the input image is resized to 640×480 at the beginning, and the color is also inverted, converting the light-colored background to dark.

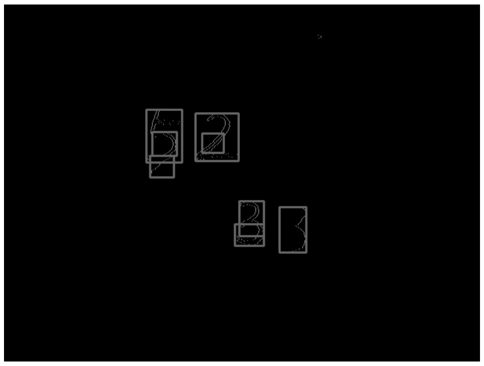
Next, we threshold pixels inside the bounding boxes using the threshold value computed according to the following equation:

$$\text{threshold} = \text{mean} + \text{standard derivation}/2 \tag{1}$$

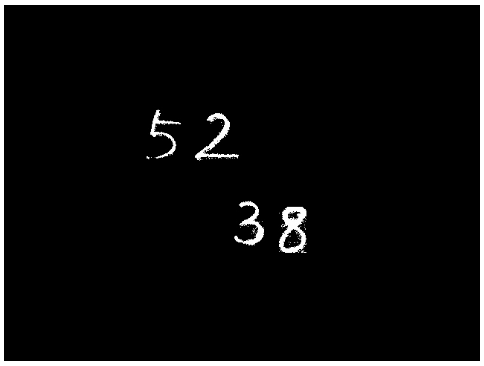
To ensure the threshold value is independent of the margin size, the mean and standard derivation is computed by all the pixels inside the bounding boxes rather than all pixels. We found the heuristic works effectively with different testing environments. The image after preprocessing is shown in Figure 1(c).



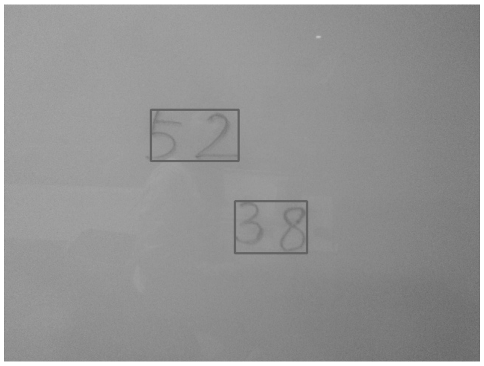
(a) Example test image



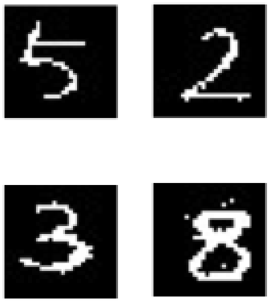
(b) Bounding box of Canny edge features



(c) Image after preprocessing



(d) Bounding box of numbers to be segmented



(e) Segmented digit patch

Figure 1: Input and intermediate image in preprocess and segmentation steps

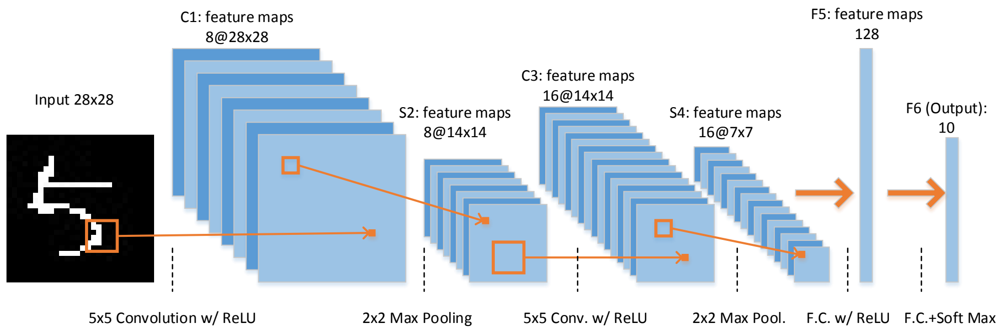


Figure 2: CNN Architecture

3.3 Segment digit patch

Even with image resized to 640×480, the image is still too large for recognition. Besides, users may want to write multiple numbers on the same page, it is useful to figure out which digits consist each number in a single pass. Therefore, segmentation step is introduced to the pipeline to resolve both issues.

We segment the image in two steps, first to find the bounding box for each number, next to segment each digit inside the bounding box. In the first step, we use the contour finder to locate each digit and draw bounding box around each digit, then by computing and comparing the positions of digits, we merge bounding boxes of digits that belong to the same number. The result is shown in Figure 1(d). In the second step, we scan through the merged bounding box from left to right, using the space (empty column) between each digit, we segment the digit patch. The digit patch is resized to 28×28 so it is compatible with the CNN input size. The segmented digit patch is presented in Figure 1(e).

3.4 Digit recognition using CNN

After the digit segmentation, the original image is slided and rescaled into $28{\times}28$ image patches of individual digits. The patches are fed into a CNN also for recognition. We use a custom CNN architecture, consisting of two convolutional layers (C1 and C3), two max pooling layers (S2 and S4) and two fully-connected layers (F5 and F6), as depicted in Figure 2. The output of F2 is passed to a 10-way softmax, which produces a probability distribution over ten labels (i.e. ‘0’-‘9’).

The first convolutional layer (C1) filters the $28{\times}28$ input grayscale image with 8 kernels of size $5{\times}5$, while the second convolutional layer (C3) filters the down-sampled $14{\times}14{\times}8$ feature maps with 16 kernels of size $5{\times}5{\times}8$. Both convolutional layers use a unit stride. Down-sampling occurs at layer S2 and S4 by applying $2{\times}2$ non-overlapping max pooling. Finally, the two fully-connected layers, F5 and F6, have 128 and 10 neurons, respectively.

The dimensions (e.g convolution windows size, layer counts, kernel counts, etc.) of the network are similar to LeNet-5 [12], which is an early CNN for hand-written digit recognition, though we use one less fully-connected layer. However, we use simpler yet more popular components to construct the network. For example, ReLU nonlinear function [15] is used at the output of the convolutional and fully-connected layers, which is much efficient to compute than the sigmoid or the hyperbolic tangent functions and also proved to be trained faster for the same accuracy [9].

3.4.1 Offline training

We build and train the CNN architecture shown in Figure 2 offline using Python. We use MNIST [10] as our training dataset. After converted from the big endian format using MATLAB, each input image is a 28×28 digit patch with grey background and white digit. We compute the mean image, and subtract each image with the mean image to form our final input patch. As the input patch is

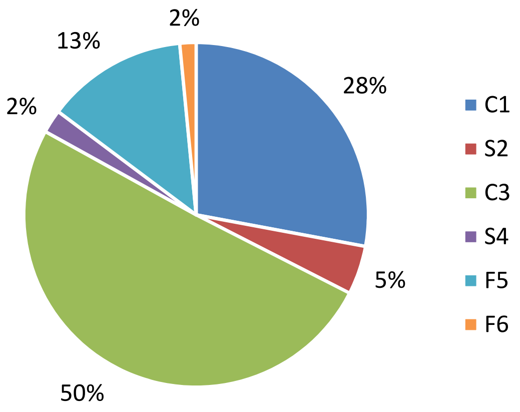


Figure 3: CNN layer runtime breakdown on x86 with SSE. The most computation intensive layers are the convolutional layers (C1 and C3) and the first fully-connected layer (F5).

single channel with only one object in the center, we did not perform any data augmentation on it. To speedup the training process and to reduce overfitting, we insert a dropout layer between F5 and F6 layer, with dropout rate 0.5. With setting the initial point as 0.01, and batching 100 images, we explore different learning rate and regularization. We found at learning rate $5e^{-4}$, regularization $5e^{-3}$, the best model can achieve 99.07% validation accuracy after training 5 epochs.

3.4.2 Mobile implementation

Implementing CNN on mobile platform can be challenging given the relatively low CPU performance and the limited memory resources. In this project, we build our CNN based on DeepBeliefSDK [16], an open source CNN framework for mobile platforms. DeepBeliefSDK is implemented in highly optimized C++ code, where convolution operations are transformed to general matrix-matrix multiplications (GEMM), with support for several GEMM libraries. For android platform, it makes use of Eigen library [17] for the NEON-optimized GEMM implementation.

DeepBeliefSDK is originally built for AlexNet [9], but the modularity of the framework allows us to largely re-use the code. Since our CNN use the same layer components (i.e. convolutional, fully-connected, ReLU, max pooling and softmax layers) as AlexNet, we construct the network manually by calling the internal functions and the class methods in DeepBeliefSDK. We later save the network in DeepBeliefSDK's standard file format, which allows our main application to use the framework through DeepBeliefSDK library API. Since our CNN is relatively shallow and narrow, we don't compress our network parameters (mostly 32-bit float weights) but dump them directly into a binary file, which ends up to be 426 kB.

For the baseline CNN implementation, besides some modifications to adapt the input image size, we use the same library API originally defined in DeepBeliefSDK. We first compiled the DeepBeliefSDK library together with Eigen library on x86 platform with multi-threading disabled. Figure 3 shows the CPU time breakdown of each layers of the CNN. The most computation intensive layers are the convolutional layers (C1 and C3) and the first fully-connected layer (F5), which consume 28%, 50%, and 13% of the total runtime, respectively.

3.4.3 Batching CNN

The last few layers of a CNN (F5 and F6 in our CNN) are usually fully connected layers, which are often implemented as a vector-matrix multiplication (e.g. in DeepBeliefSDK). While this is optimal in terms of the latency of recognizing a single image, it is inefficient for our multiple image recognition tasks. By batching the computation of the fully connected layer for several images, we can

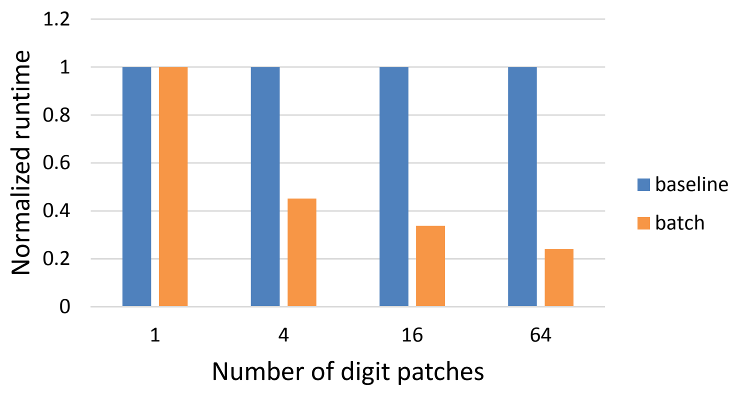


Figure 4: CPU runtime of the fully connected layer F5 batching 1, 4, 16, and 64 images. The runtime is normalized by the corresponding runtime of the baseline implementation. The more images batched, the greater speedup is achieved.

replace several vector-matrix multiplications with the more efficient matrix-matrix multiplication, which can take advantage of CPU caching the layer weights [14].

Figure 4 illustrates the benefit of batching fully-connected layer computation. Generally, the more images batched at the fully-connected layer, the greater speedup is achieved through batching. For our CNN architecture, batching 4, 16, and 64 images at layer F5 reduce the total runtime of the F5 computation by 55%, 66% and 77%, respectively, compared to the baseline. We implement this batch mode execution in DeepBeliefSDK as an option for running CNN.

4 Experiments

4.1 Build method

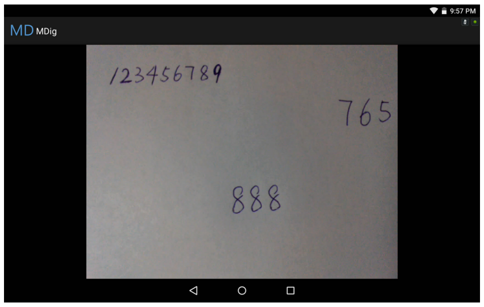
The main image processing pipeline, including preprocessing, segmentation and CNN, is implemented in C++ and built using Android NDK 10d [18], while the application UI uses Android SDK API 21 [19] and OpenCV java library [20]. The C++ code depends on two libraries, OpenCV and our modified DeepBeliefSDK. We use a prebuilt OpenCV 2.4 library from Tegra Android Development Pack [21], which is optimized for NVIDIA Tegra devices including our target device, NVIDIA SHIELD tablet. We build the DeepBeliefSDK with Eigen library using NDK with NEON optimization. We turn off multi-threading optimization in Eigen library for more consistent performance analysis.

Our project source code is available at `https://github.com/jingpu/MDig`.

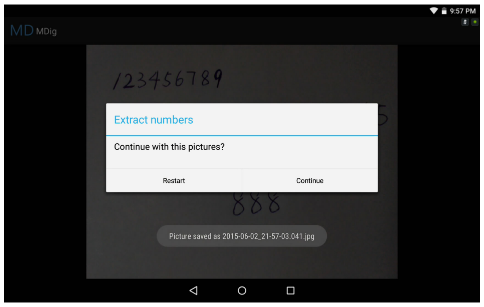
4.2 Android UI demonstration

After entering the application, user can take a photo of any hand-written numbers (Figure 5(a)). If the user confirm to continue digit recognition with the taken picture (Figure 5(b)), the result will appear in the next dialog box (Figure 5(c)).

A video demo is available at `https://youtu.be/X_cULsdAH1o`. The video consists of a set of three tests, which demonstrates MDig's ability to recognize 10 different digits and multiple number strings, and the robustness over different types of pens.



(a) Capture mode



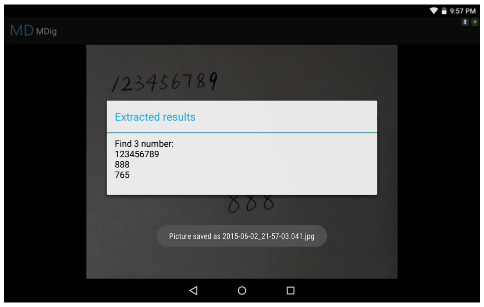
Extract numbers

Continue with this pictures?

Restart | Continue

Picture saved as 2015-06-02_21-57-03.041.jpg

(b) Confirmation screen



Extracted results

Find 3 number:
123456789
888
765

Picture saved as 2015-06-02_21-57-03.041.jpg

(c) Recognition result

Figure 5: Screenshots of MDig Android UI.

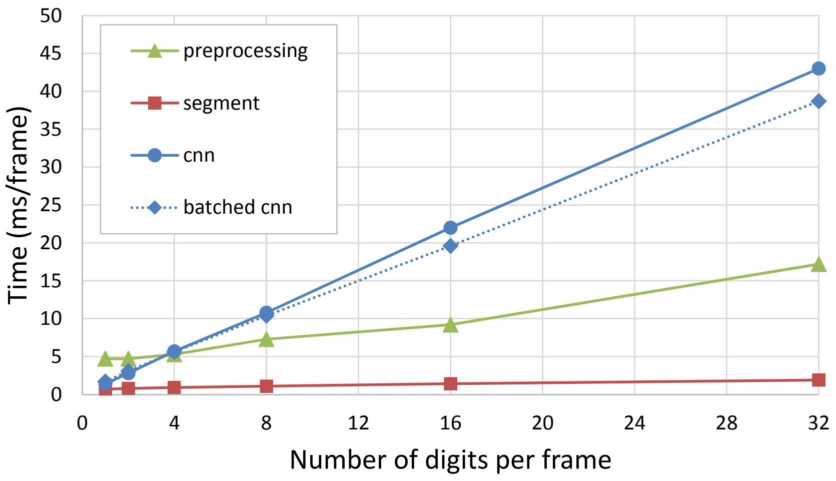


Figure 6: Runtime breakdown between stages with variant number of digits per frame.

4.3 Performance analysis

For the performance analysis, we prepare a set of pre-taken pictures containing 1, 2, 4, 8, 16 and 32 digits. Then we feed each picture to the image processing pipeline of MDig on SHIELD tablet with the camera funtion disabled. We measure the pre-process CPU time for several time, and take the average.

Figure 6 illustrates the runtime of each stage of the pipeline scaled with the number of digits in the frame. The CNN stage runtime scales linearly proportional to the number of digits in the frame, since the CNN is called once for each digit in the baseline implementation. The preprocessing stage scales slowly with the number of the digits because the Canny edge detection is a per-frame operator and runs for about 5ms. The segmentation stage is very efficient and only operates on small bounding boxes, so it runs much faster compared to the other two stages.

Figure 6 also shows the performance of the batched CNN stage, in which the intermediate feature maps for each input image are batched before the the fully-connected layer F5 and F6, and the fully-connected layers run as decribed in Section 3.4.3. For the frame containing 32 digits, the batched CNN is 12% faster than the baseline CNN. The speedup of running CNN in batch mode is limited by the ratio of the fully-connected layer compution to the total compution, since the convolutional layers run exactly the same. In our case, the portion of the fully-connected layer computation is 15% as shown in Figure 3. We expect the batching optimization to be more effective for any CNNs using relative larger fully-connected layers.

5 Conclusions

In this project, we demonstrate a mobile application, MDig, which recognizes multiple digits and number strings in the frame using a custom convolutional neural network. To implement the CNN on the mobile platform, we consciously design a shallow and narrow network, which still achieves 99.07% validation accuracy on MNIST dataset after carefully training. We apply two additional optimizations, digit segmentation and batched fully-connected layer of CNN, to further improve the real time performance. On the target device, NVIDIA SHIELD tablet, MDig processes a frame and extract 32 digits in about 60ms. The batched CNN optimization reduce the CNN runtimes by 12%.

References

[1] Kevin Jarrett, Koray Kavukcuoglu, Marc'Aurelio Ranzato, and Yann LeCun. What is the best multi-stage architecture for object recognition? In *Computer Vision, 2009 IEEE 12th International Conference on*, pages 2146–2153. IEEE, 2009.

[2] Dan Ciresan, Ueli Meier, and Jürgen Schmidhuber. Multi-column deep neural networks for image classification. In *Computer Vision and Pattern Recognition (CVPR), 2012 IEEE Conference on*, pages 3642–3649. IEEE, 2012.

[3] Marc Aurelio Ranzato, Fu Jie Huang, Y-Lan Boureau, and Yann LeCun. Unsupervised learning of invariant feature hierarchies with applications to object recognition. In *Computer Vision and Pattern Recognition, 2007. CVPR'07. IEEE Conference on*, pages 1–8. IEEE, 2007.

[4] Christopher Poultney, Sumit Chopra, Yann L Cun, et al. Efficient learning of sparse representations with an energy-based model. In *Advances in neural information processing systems*, pages 1137–1144, 2006.

[5] 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.

[6] Parul Shah, Sunil Karamchandani, Taskeen Nadkar, Nikita Gulechha, Kaushik Koli, and Ketan Lad. Ocr-based chassis-number recognition using artificial neural networks. In *Vehicular Electronics and Safety (ICVES), 2009 IEEE International Conference on*, pages 31–34. IEEE, 2009.

[7] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[8] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. *arXiv preprint arXiv:1409.4842*, 2014.

[9] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C.J.C. Burges, L. Bottou, and K.Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012.

[10] Yann LeCun, Corinna Cortes, and Christopher J.C. Burges. The mnist database of handwritten digits. `http://yann.lecun.com/exdb/mnist/`.

[11] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. *arXiv preprint arXiv:1502.01852*, 2015.

[12] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[13] Alex Graves et al. *Supervised sequence labelling with recurrent neural networks*, volume 385. Springer, 2012.

[14] Vincent Vanhoucke, Andrew Senior, and Mark Z Mao. Improving the speed of neural networks on cpus. In *Proc. Deep Learning and Unsupervised Feature Learning NIPS Workshop*, 2011.

[15] Vinod Nair and Geoffrey E Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, pages 807–814, 2010.

[16] Jetpac. Deep belief sdk. `https://github.com/jetpacapp/DeepBeliefSDK`.

[17] Gaël Guennebaud, Benoît Jacob, et al. Eigen v3. `http://eigen.tuxfamily.org`, 2010.

[18] Google Inc. Android ndk. `https://developer.android.com/ndk`.

[19] Google Inc. Android sdk. `https://developer.android.com/sdk`.

[20] Itseez. Opencv. `http://opencv.org/`.

[21] NVIDIA Corporation. Nvidia androidworks. `https://developer.nvidia.com/AndroidWorks`.