CS230

A Multi-digit OCR System for Historical Records (Computer Vision)

Jonas Mueller-Gastell
Department of Economics
Stanford University
jonasmg@stanford.edu

Marcelo Sena
Department of Economics
Stanford University
msena@stanford.edu

Chiin-Zhe Tan
Stanford Center for Professional Development
Stanford University
chiin@stanford.edu

1 Introduction

We study handwritten multi-digit number character recognition in historic census documents. To the best of our knowledge there are no algorithms that have human-comparable performance for this task. The end goal is to deploy a product which has two distinct features. 1) A preprocessesing step adapted to the specific data set that we wish to digitize, 2) a generally useful deep learning architecture to digitize handwritten multi-charcater strings. The preprocessing step will serve to extract cells from a given table scan (a task that in principle could also be performed by a generalist deep learning algorithm, but likely at a loss of performance vis-a-vis a human engineered specialized product for our data). Previous related attempts of digitizing hand-written historical documents include de Buy Wenniger et al. (2019) and Wigington et al. (2018), both specialized for long-form text rather than tabular numeric information.

2 Dataset

Our data set contains images from the 1940 agricultural census records of Finland. We were in the process of scanning the approximately 250,000 records available when COVID-19 lockdown orders closed the archives. Even with only 28 images available, we already have a sufficient sample to develop simple algorithms. We have extracted 8308 cells from these images, of which 1303 are blank and 4385 contain a placeholder symbol ('-'). The figures describing the data distribution of the remaining 2620 cells in greater detail are found in the Appendix (Figures 2a, 2b). As expected, there is a large number of 0s, 1s, and 5s, but also a small number of special characters (',', '/', '+', '?').

Given the small data set size, we forgo a proper test set and use 75% of the data as train and 25% as dev set. Once COVID19 restrictions ease in Finland, we will perform a real test-set evaluation on completely new data. We take care to sort the data such that train and dev set come from different pictures. This helps us gauge the true out of sample loss we should expect when digitizing new pages, as the handwriting of different authors is likely different.

We try to overcome the limitations of this small training set in three ways: 1) we employ a variety of data augmentation steps on the actual training examples; 2) we have created synthetic multicharacter strings similar in features to the original data, by composing multiple images from the MNIST database (LeCun and Cortes, 2010). We employ this data in two different ways: a) we train the full

CS230: Deep Learning, Spring 2020, Stanford University, CA. (LateX template borrowed from NIPS 2017.)

model on the synthetic data and then freeze the lower layers and retrain the upper layers at a reduced learning rate using the real data; b) we mix the train set of the real data and the synthetic training data, shuffle, and train the entire model, while using only the real data as validation set. 3) the last method we experiment with to overcome our small dataset restriction is transfer learning from pretrained models. We use VGG-19 (Simonyan and Zisserman, 2014) and ResNet-50 (He et al., 2016) en lieu of our own convolutional stack and only train the upper recurrent layers on our real data.

3 Pipeline Overview and Network Architectures

All code can be found in our GitHub at `https://github.com/JonasEc/CS230AWS`. The preliminary overall program architecture is as follows:

1. We preprocess the original census scan to center all images identically and place a grid with hard-coded cell boundaries over the original census image and extract the thus bounded cells. See Figure 3a in the Appendix for an example.

2. We preprocess the cell images by eliminating page background color, turning to grayscale, and normalizing the range of contrast. See Figure 3b in the Appendix for an example. We then standardize cell-image height and width to (64, 128).

3. A simple 8-layer convolutional neural net (CNN) determines whether or not the cell is contains a non-empty, non-placeholder string, using a binary crossentropy loss. As we already achieve 98% accuracy on the dev set with this simple model, we have concentrated efforts on improving the pipeline elsewhere.

4. The actually populated cells are fed through a deep multi-stage network: we compose a CNN-Dense-RNN and train it using the Connectionist Temporal Classification (CTC) loss, in order to translate a sequence of image features into a sequence of numbers.

   (a) A 6 layer CNN extracts features from the image and scales it down to 1/8th its original size

   (b) A max-pooling layer with asymmetric kernel and a single dense layer translate the image features into 16 time steps and 128 features

   (c) A two layer, bidirectional gated recurrent unit RNN processes the features and a final softmax layer makes predictions to be fed through the CTC loss

The basic architecture of the CNN-RNN-CTC is displayed in Figure 1 (generated using Bäuerle et al. (2019)). This is a rather parsimonious model: the convolutional step doubles features after each max pool, topping out at 256 features, and the RNN layers only use 128 neurons in each direction. In total, we use just over 1.7 million parameters. Nonetheless, we achieve 86.1% accuracy on the real data dev set when using no data augmentation or pre-training. On the synthetic data we achieve 97.0% accuracy. This architecture is loosely based on `https://keras.io/examples/image_ocr/`.

We also experimented with larger architectures. The largest architecture that seemed to gain some performance on synthetic data replaces the 6 layer convolutional stack with 10 such layers, with the final layer having 1024 rather than 256 features. We also scaled up the dense and GRU neuron count to 256 each. Overall, this increases parameter count to over 21 million. This raises accuracy on the synthetic dev set to 97.6%, only a modest improvement. Moreover, transfer learning from the synthetic data to the real data did not yield promising results, so we abandoned this avenue for now.

What performance should we expect from these models? A research scientist with subject matter expertise who reads the original forms will likely make near zero mistakes, except perhaps data-entry (key stroke) errors from fatigue! Thus, unavoidable bias is likely near 0. However, one should also evaluate performance versus a realistic baseline: data entry by non-experts on platforms such as labelbox, MTurk, or by data entry firms situated in India. The error rate for these workers is likely in the range of 0.5-2%.

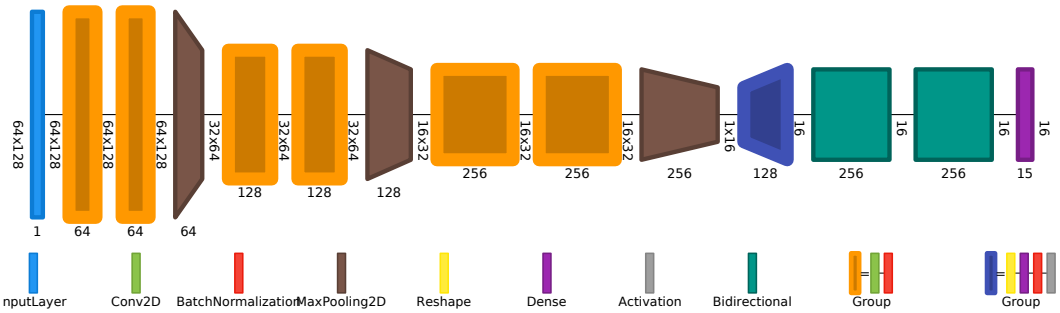


Figure 1: Architecture of the CNN-RNN-CTC. Each orange layer composes a 2D convolution, batch norm, and RELU activation. Kernel sizes are (7,7) and (5,5) for the first two layers and (3,3) for all subsequent layers. To translate from (2D $\times$ Features) to (timesteps $\times$ features), we use pooling with kernel (16,2) and a single dense layer with 128 neurons, batch norm, and RELU activation. The bidirectional layers use concatenation of 128 GRU units with 0.5 dropout. The output sequences are fed into a 15 output neurons softmax activation layer.

4 Training the Models: training design, data augmentation, and synthetic data

We first discuss the basic design philosophy for *trainability* and *overfitting prevention*, and then explain how we try to *overcome the small training set size*. We then discuss the actual optimization progress and results.

Trainability: Even our smallest model consists of 10 layers in total, and we wanted to experiment with much deeper models. Thus, we optimized our network architecture to prevent vanishing and exploding gradient problems and ensure speedy convergence: 1) we use batch normalization for each convolutional and the dense layer, 2) we experiment with a variety of optimizers and learning rate schedulers – in the end we found a simple SGD with momentum and constant learning rate decay to outperform ADAM in most cases; in addition, we set an automatic learning rate reduction by factor $\frac{1}{4}$ whenever a validation loss plateau was encountered. For speedy training, we use an AWS EC2 instance of type "p3.2 large".

Overfitting prevention: The basic data set only leaves 1965 training examples after the train-dev split. We thus were prone to overfitting. To lessen this risk, we use aggressive dropout in the bidirectional GRU layers with a 50% dropout rate, aggressive max pooling, and automatic early stopping of the learning algorithm when the validation loss no longer improves. In addition, we experiment with data augmentation, detailed below.

These elements combined yield promising performance on their own. Training stops on early-stopping after 42 epochs and achieves a training loss of 0.0978 and accuracy of 0.9585. Validation loss is higher at 0.9211 and validation accuracy is a little lower at 0.8609, indicating some degree of overfitting. Results from all training variants are displayed in Table 1.

4.1 Data Augmentation

Given the small data set and the presence of overfitting, we employ data augmentation. In a separate training run, we distort each element of each batch with the following steps:

1. Random brightness distortion, uniformly increasing or decreasing brightness by up to 20%,
2. Addition of noise pixels (uniform random intensity speckles) in up to 8 random locations and deletion of information in up to 8 random (4,4) pixel areas,
3. Addition of bounding lines to any or all of the four sides of the picture (to mimic the presence of such lines when the cell extraction is off by a couple pixels),
4. Rotation of the picture by random amount between -30 and $+30$ degrees.

Examples of these operations are found in Figure 4 in the Appendix. Combining these features slows training down considerably but has some positive effect. Training loss: 0.4538, training accuracy:

0.8289, validation loss: 0.5781, and validation accuracy improves by more than 1% to 0.8734. **Note that training loss and accuracy are no longer comparable to before, as these pictures are now considerably harder to read. Thus, it is unsurprising that training set accuracy falls when using data augmentation. Often, we have lower training accuracy than dev accuracy when employing data augmentation, as the we do not augment the dev set data.** We experimented with decreasing the learning rate for the data augmented models and prolonging training by allowing more patience for learning rate reductions on validation loss plateaus and early stopping, with limited impact on validation accuracy.

4.2 Generating Synthetic Data

The synthetic data is built up from MNIST digits and we aim to mimic the visual features of the real data as much as possible. The algorithm for creating a synthetic example is detailed in the Appendix and example images are also in the Appendix in Figure 5. We generate $75,000$ training examples and $5,000$ dev set examples. We then employ two different strategies to make use of this synthetic data. First, we use transfer learning from a model trained solely on the synthetic data. Second, we mix synthetic and real data in the training set.

4.3 Transfer Learning from Synthetic Data

First, we train the simple model described above until early stopping on exclusively synthetic data (i.e., both for training and dev set). We achieve 97% accuracy on the synthetic dev set. We then proceed to use transfer learning to adapt the trained model to the real data. To do this, we load the weights of the model trained on synthetic data and set the entire CNN stack to be non-trainable. We leave the softmax layer, the bidirectional GRUs, and the dense reshaping layer trainable – this means about $350,000$ out of 1.7 million parameters can be retrained. Further, we reduce the starting learning rate by a factor 10x to 0.0005, to reduce the risk of destroying the carefully trained weights.

We initially experimented with leaving all layers except the softmax layer untrainable but the just under 4000 weights of the softmax layer were not capable of adapting to the GRU output sufficiently flexibly. One theoretical reason for this is that the GRU layer can and should learn the non-random sequencing of numbers. In the synthetic data, numbers are fully random – but in reality, 1s, 5s, and 0s are more common, and a comma is often followed by a 0, for example. Thus, the GRU layers need to learn from the real dependency in the data for maximal efficiency. In the future, we will explore adding this dependency to the synthetic data, too.

Unfortunately, transfer learning does not seem to yield large benefits – in fact, we have slightly smaller dev set accuracy than without transfer learning! Without further data augmentation, we only reach 0.8453 dev accuracy, and with data augmentation we actually fall to 0.8406 dev accuracy. Interestingly, we performed slightly better using ADAM rather than SGD with momentum when using transfer learning, reversing our earlier experience.

4.4 Mixing Synthetic Data with Real Data

We use the full $75,000$ synthetic examples shuffled into the real data training set and train the simple network on this combined data set. We keep the dev set purely real data. This has two reasons: 1) we want to know what the real accuracy on real data is; 2) we want to use early stopping and learning rate scheduling based on feedback off the real accuracy.

This approach seems very promising – significantly more so than using synthetic data and employing transfer learning afterwards. Without data augmentation, we achieve a dev-set accuracy of 0.9031. When employing data augmentation, we used several runs of the optimization procedure, experimenting with the patience of early stopping and learning rate scheduling. The best run achieved a similar dev accuracy of 0.8953. It seems the data augmentation process does not drastically improve performance once data is plentiful.

5 Transfer Learning from Pre-Trained Networks: VGG19 and ResNet50

We attempted transfer learning from well known pretrained convolutional neural networks, namely VGG-19 (Simonyan and Zisserman, 2014) and ResNet-50 (He et al., 2016). We removed the

	Training		Validation	
	Loss	Accuracy	Loss	Accuracy
Small network w/o Data Augment	0.0978	0.9585	0.9211	0.8609
Small network with Data Augment	0.4538	0.8289	0.5781	0.8734
Small network w/o Data Augment, Transfer learning from synthetic	0.7862	0.7049	0.6187	0.8453
Small network with Data Augment, Transfer learning from synthetic	1.6187	0.4892	0.6499	0.8406
Small network, mixing Real & Synth	0.0176	0.9950	0.5848	0.9031
Small network, mixing Real & Synth, with Data Augment	0.2534	0.9150	0.5467	0.8953
Small network, mixing Real & Synth, with Data Augment, higher patience	0.1609	0.9456	0.5208	0.8781
Big network w/o Data Augment, Transfer learning from synthetic	0.9372	0.6619	1.1915	0.7078
Transfer learning from VGG-19	2.27	0.64	2.35	0.70
Transfer learning from ResNet-50	10.66	0.34	8.30	0.69

Table 1: Loss and Accuracy for the main models studied

convolutional layers used in the baseline model described above and added instead a pretrained ResNet-50 or VGG-19 before our GRU layers. These layers are always frozen, since one of the main motivations for this exercise is exactly to overcome our main hurdle which is our small training dataset. We experimented adding different number of frozen layers and settled with 15 layers from VGG19 and 50 layers of ResNet50. The pre-trained models all operate on RGB images while we make classification on gray-scale images. To adapt the dataset, we replicate our single channel 3 times to make it conformable with the above networks.

Both ResNet50 and VGG19 are both readily available in `Keras.applications`. It seems likely that other models could have achieved higher validation accuracy. Both VGG-19 and ResNet-50 are trained on ImageNet data, which is not ideal for our objective. A model trained on MNIST, for example, would potentially push our accuracy even further. The lack of many such models is also the explicit reason for *why* we undertook this project: as of yet, there are few papers exploring digitization of multi-digit strings from handwritten documents; even fewer of these papers have code implementations available online and none that we could find had pretrained weights available for download. We hope that with our project we can eventually help to rectify this lack of available models with weights.

Our general takeaways from this attempt is that using pretrained convolutional networks seems promising in general. Exploring different setups with both pretrained neural networks, we managed to obtain accuracy of 70% on the validation set, with very short training time. While this is inferior to our other transfer learning attempts, they are showing that even models trained on vastly different data produce weights of some use to our algorithm.

6 Future Work

While CS230 is now over, we do plan to continue to iterate on this project. Much more hyperparamter tuning and network architecture experimentation remain to be done. More training data will soon be available. However, we also plan to experiment with more fundamental changes to our approach. Fundamentally, we want to generate a table digitization software. Tables have inter-cell dependency. The CNN-RNN-CTC architecture allows us to exploit the non-randomness of multiple consecutive digits. A human readers exploits the inter-cell dependency, too. To mimic this behavior, we will experiment with end-to-end learning several cells at once, with the CNN-RNN for each cell embedded in a larger RNN that uses an entire row or column of cells as input.

Contributions

We discussed the general angle of attack on the problem, the methods to overcome our small training size (data augmentation, transfer learning, synthetic data), high-level architecture choices, and the write up of the proposal, milestones, and final reports jointly. No demarcated contribution assignment is feasible for these high level features of our project. Below we discuss each team member's specific detailed assignments.

Jonas Mueller-Gastell: Jonas was in charge of the nitty-gritty of the architecture implementation and training on AWS. Jonas experimented with hyperparameters such as network size and shape, regularization techniques, learning rate schedules, etc. Jonas merged and fine-tuned the work performed by Chiin and Marcelo on their respective tasks into the main programming pipeline. Jonas also took Chiin's synthetic data generation code and refactored much of into data augmentation methods.

Chiin-Zhe Tan: Chiin was leading the project on synthetic data generation. He developed the algorithm for turning MNIST data into life-like fake data, first using OpenCV2 and then refactoring as much as possible into TensorFlow based code. Chiin was also the main "bugtester", sanity checking the main architecture pipeline by re-implemeting it offline on his machine – both to find programming bugs as well as to ensure our program made not just syntactic sense but did what we wanted it to do.

Marcelo Moura Jardim Teixeira Sena: Marcelo was in charge of implementing the transfer learning routines from pretrained models. Jonas and Marcelo discussed and planned implementing the different methods for transfer learning in the project, and split up the tasks, with Jonas implementing transfer learning from synthetic data and Marcelo implementing transfer learning from pretrained models. He also took charge of the alterations to the model architecture necessitated by pasting the pretrained deep networks to our existing recurrent architecture.

References

Bäuerle, A., van Onzenoodt, C., and Ropinski, T. (2019). Net2vis – a visual grammar for automatically generating publication-ready cnn architecture visualizations.

de Buy Wenniger, G. M., Schomaker, L., and Way, A. (2019). No padding please: Efficient neural handwriting recognition. *CoRR*, abs/1902.11208.

He, K., Zhang, X., Ren, S., and Sun, J. (2016). Deep residual learning for image recognition. *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

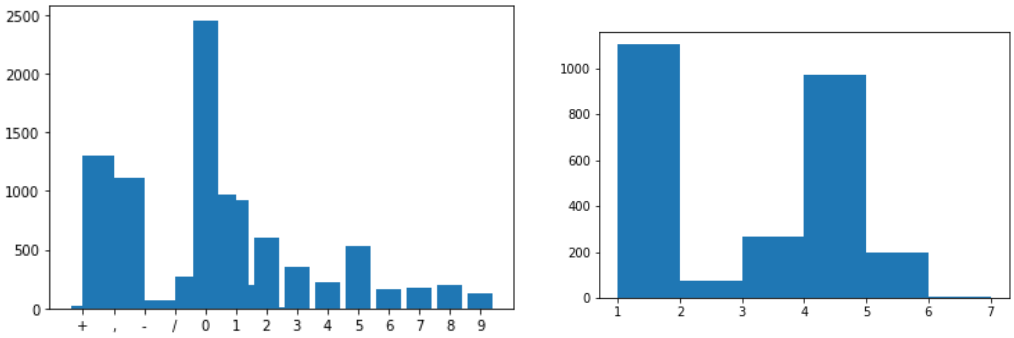
LeCun, Y. and Cortes, C. (2010). MNIST handwritten digit database.

Simonyan, K. and Zisserman, A. (2014). Very deep convolutional networks for large-scale image recognition.

Wigington, C., Tensmeyer, C., Davis, B., Barrett, W., Price, B., and Cohen, S. (2018). Start, follow, read: End-to-end full-page handwriting recognition. In *The European Conference on Computer Vision (ECCV)*.

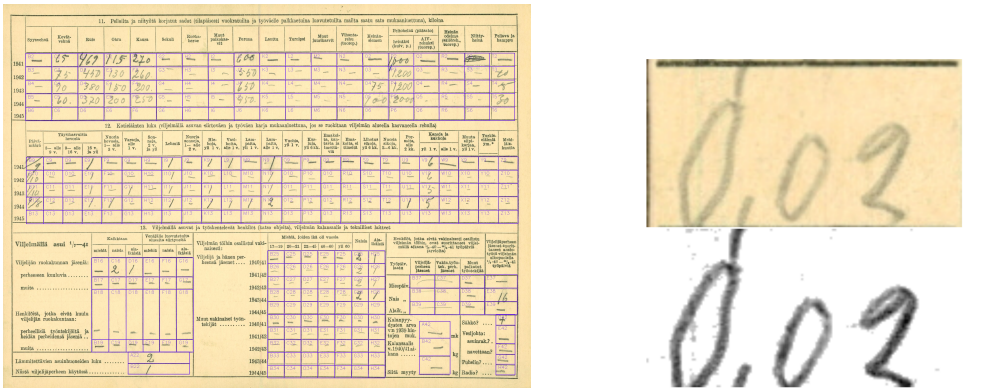
`https://keras.io/examples/image_ocr/`

A Appendix



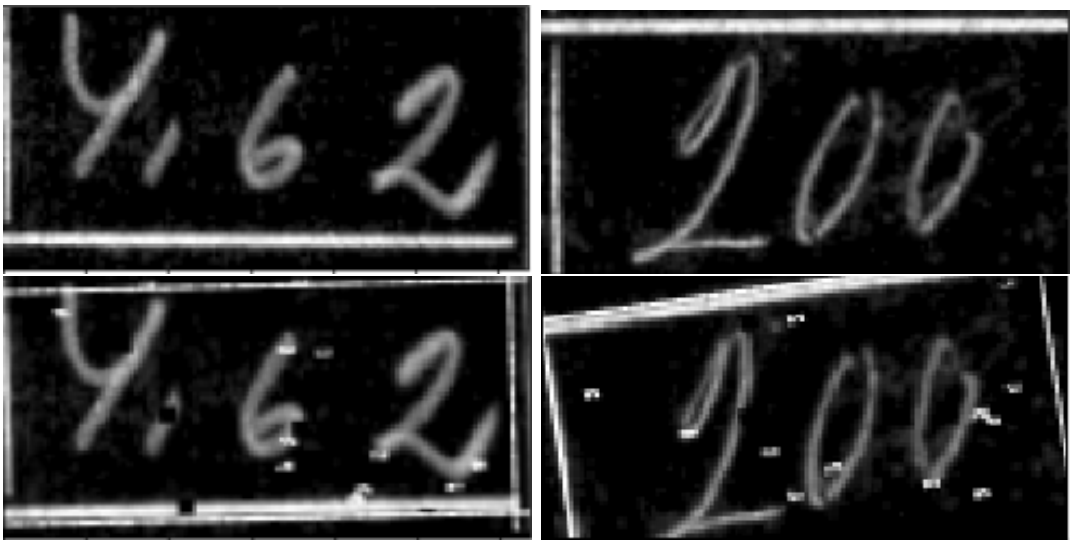
(a) Distribution of Characters (b) Length Distribution of Strings

Figure 2: Data Description



(a) Example of Grid on Scan (b) Isolated Cell ⇒ Preprocessed Cell

Figure 3: Extraction and Preprocessing



(a) Example of Line Addition and Info Deletion (b) Example of Rotation and Noise Addition

Figure 4: Example of Data Augmentation Process

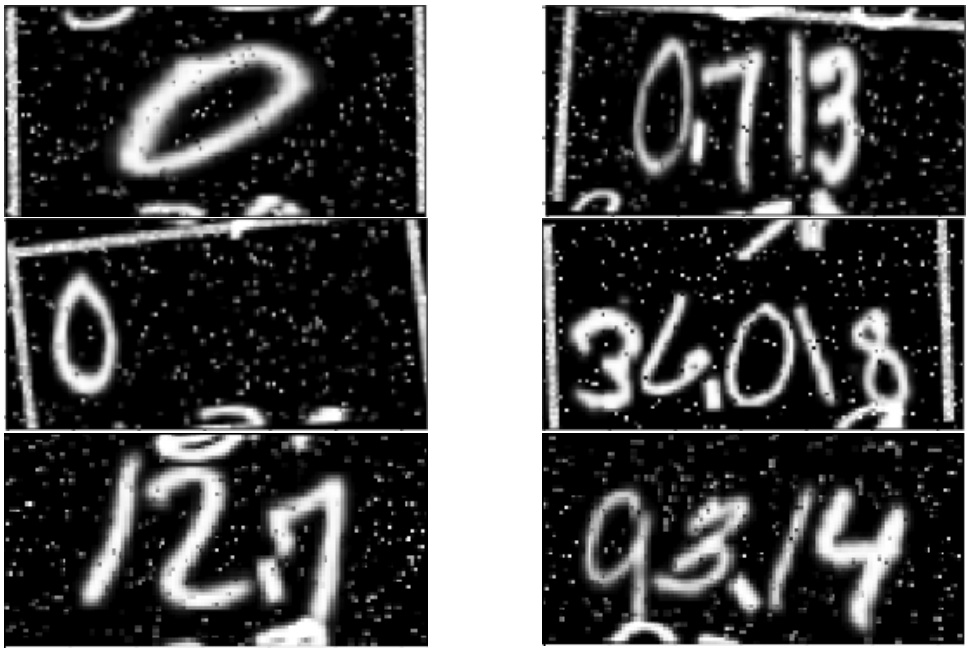


Figure 5: Example of Synthetic Data

A.1 Synthetic Data Algorithm

1. Random determination of sequence length k (up to six digits) and sampling of k random digits from the MNIST data base
2. Concatenation of the randomly rotated six digits, with some reduction in overlap between the digit images. Here, with some probability up to one "comma" character is inserted in a random position.
3. Random resizing of the resulting digit image, to ensure different scale of images
4. Random padding to the left, right, top, and bottom, to yield desired image dimension (64,128) with the resulting image having a random center location
5. Up to four additional random digits are inserted above or below the number in focus, partially outside the picture frame (to emulate overlap in the real data)
6. Random addition of pure noise
7. Random cell box lines, with some random rotation of the box lines
8. Random rotation of the composite image