Classification and Detection with Convolutional Neural Networks

Xiaodong Zhu
Xiaodong.zhu@gatech.edu

Recognizing the numbers from the real-world images is a long-standing challenge in computer vision field. In this report, I trained a convolution neural network by using the street view house numbers (SVHN) dataset. **Methods.** I then apply this detector to recognize the house numbers from the real-world images. I have compared the performance of the different models, including (1) a small CNN with only 2 convolution layers constructs by myself, (2) a CNN with 5 convolution layers, (3-5) VGG16 model with the pretrained weights freeze, and with 2 additional dense layers of 128,512,1024 nodes respectively. (6) VGG16 model with the all pretrained weights trainable and 2 additional dense layers of 1024 nodes. (7) VGG16 model without any pretrained weights and trained from scratch. I have also tested the effects of learning rates and batch sizes on the model. **Conclusion.** VGG16 with 2*dense(1024), and the freezed pretrained weights gives me the best performance. On the SVHN testing set, I have the average accuracy per digits: 0.945. The full number recognition accuracy is 0.826

Part1. Build the convolution Neural Networks

SVHN preprocessing

Dataset: SVHN dataset composed of the real word images from Google Street View images [1]. This dataset has been previous explored by various of methods, such as SVM [2], Convolution network, with different architectures [2-3]. In particular I have referred several [4-5] for my implementation.

There are two formats of data in the SVHN, one is for single digit, one for multiple digits. *I* choose to use the one with multiple digits, so my learner directly detects multiple digits all at once. Thus, SVHN hereinafter referred to its multidigit format. Moreover, In the SVHN, very few data have 5 digits. In order to make sure that all the digits can have enough training samples, I decided to focus on numbers with 4 digits or less. The SVHN data has 3 sets, I only used test set and training set, due to the computational power of my computer. I have split training set for two sub group one for validation, one group is for training. Thus my final data sets are: Test set contains 12920 samples. Training set contains 22370 samples. Validation sets contains 9588. Each sample has no more than 4 digits

Preprocess: In SVHN data set, the bounding box of the numbers was provided. I have enlarged the bounding boxes of 20%, and then crop the images inside of those boxes and use it for training. I have also performed a standard normalization of the image data. The input images were then also resized to 32*32. The processed data and the label are stored in h5 files.

Model Selection

Model selection. To compare the model, I have fixed batch size=128, learning rate=0.0001, optimizer=adam, and then compared different network architecture

(1). For VGG16 model with the locked pretrained weights, and only train the last 11 layers. I have tested 3 different architectures below:

VGG model_1: VGG16 output→ average pool→ 2 dense (128)→ 4 dense(11)
VGG model_2: VGG16 output→ average pool→ 2 dense (512)→ 4 dense(11)
VGG model_3: VGG16 output→ average pool→ 2 dense (1024)→ 4 dense(11)

(2). I then use a VGG model,and retrained all the pretrained weights.
VGG model_4: VGG16 output→ average pool→ 2 dense (1024)→ 4 dense(11)

(3). I have also construct a VGG model without pretrained weights, and trained it from scratch
VGG model_5: VGG16 output→ average pool→ 2 dense (1024)→ 4 dense(11)

(4). I have construct two of my own models, just to compare it with the VGG16
my model_1: conv2D(32)→conv2D(16)→dense(8) → 4 dense(11)
my model_2:
conv2D(32)→conv2D(64)→conv2D(128)→conv2D(256)→conv2D(512)→dense(1024)→4 dense(11) as output

Table1 | the testing performance of different CNNs on SVHN testing set

	My models		VGG 16 with the pretrained weights Either freeze the internal weights or retrain				VGG 16 no pretrained weights
	2 conv layers	5 conv layers	Freeze 2 dense (128)	Freeze 2 dense (512)	Freeze 2 dense (1024)	retrained (2 dense 1024)	Trained from scratch (2 dense 1024)
Average accuracy per digit	0.923	0.680	0.921	0.940	0.945	0.940	0.940
Full digits accuracy	0.765	0.139	0.738	0.808	0.826	0.809	0.809

As shows in the table1 above, VGG16(dense 1024),with the pretrained weights, gives the best result of average accuracy per digits=0.945, full accuracy=0.826. I have selected 4 models, and plot their training curve in the figure1, down below.

Fig1A |My model with 5 Conv layers

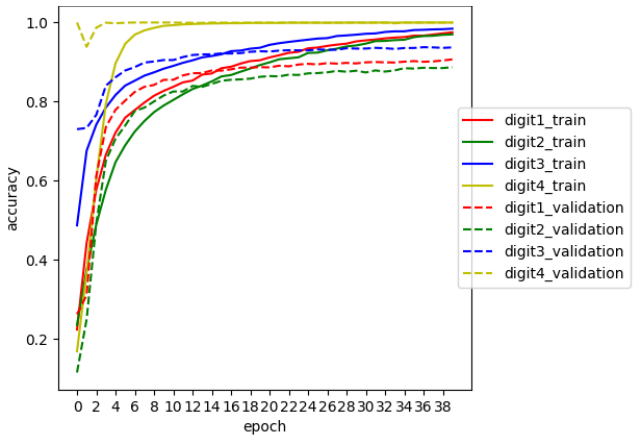


Fig1B |A VGG16 with pretrained weights locked

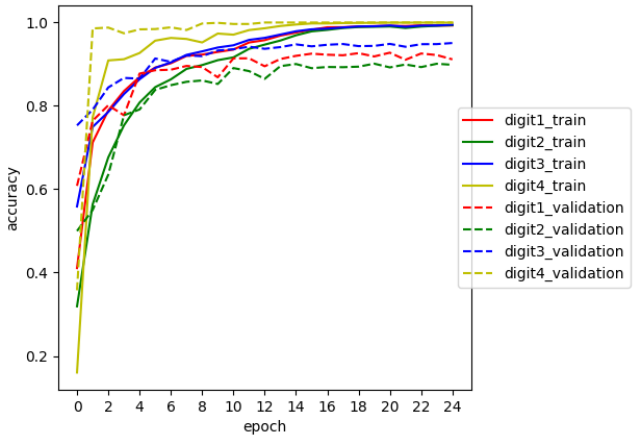


Fig1C |A VGG16 with pretrained weights retrained

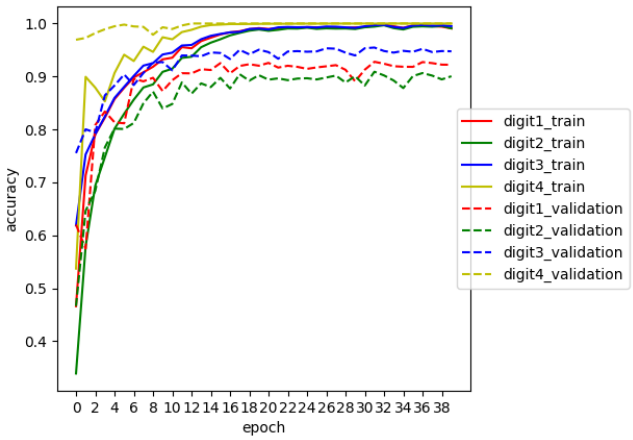
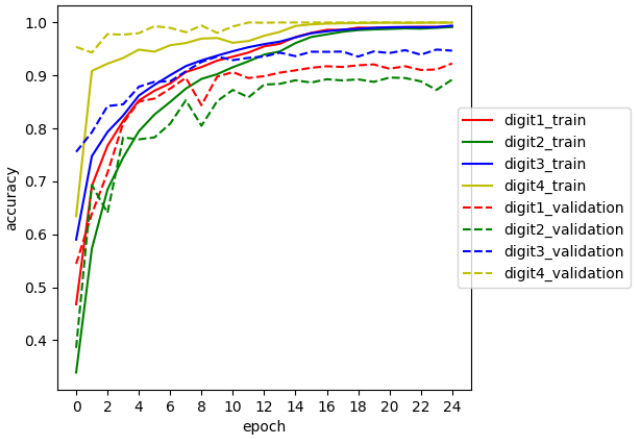


Fig1D |A VGG16 without any pretrained weights



From these plots, in all models, we have the last digits converge quickly. I think this is because that a lot of samples are less than four digits. Therefore, we have vast majority of the samples have nothing at this digit. So it converges quickly

CNN model parameters

From the above experiments, I conclude that VGG16 model with 2 additional dense (1024) layers, and the pretrained weights fixed, performs the best. So, I next further explored the network parameters using this model. Due to the limited computational power, I only focused on several parameters important for gradient descent. Moreover, I have used only 6000 samples in order to speed up the process.

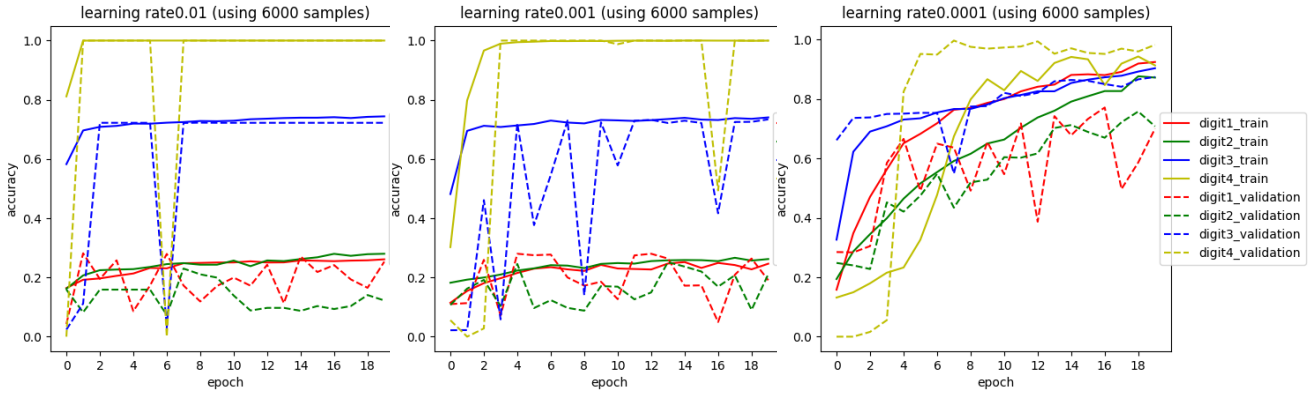
Epochs, and the training termination. Training one epoch means the entire data set has been propagate forward and backward through the network once. For gradient descent, we optimize the network in an iterative manner. So, when without enough epochs, model will be underfit. Too much epochs might leads to overfit. My solution for this: I did not early stop the training. Instead, I set epoch=25-40, and record only the best parameters during the training by using modelCheckPoint. In this case, I can avoid both under fitting and overfitting as long as I set an epoch big enough. As figure1. After around 15 epochs, the network is not improved.

Loss function: SVHN is multiclass problem. So, I used the categorial cross entropy as the loss function[6].

Optimization and learning rate. Adam is one of the most popular methods[7]. it combines the advantages of the adaptive gradient algorithm and also the Root Mean Square Propagation, therefore gives good result most of the time. Learning rate: Learning rate is how fast the weights are updated. With a larger rate, learning is faster, and fewer training epochs will needed. But with large learning rate we might be stuck at alocal optimal points. When learning rate is too small, we take too many epochs for training. Using 6000 samples. I compared the effect of the learning rate:0.01,0.001,0.0001 on the model of VGG16_model3.

As shown in the plots below. Both when learning=0.1, and 0.01, the training and validation both reach the plateau, and stays at the low accuracy and fail to improve. With learning rate=0.0001, we have the continuously improved validation and training accuracy. We can also use different strategy to decay the learning rate during the training. However, due to the computational power, I did not address such learning rate decay in this report.

Figure2| The effect of different learning rate on the model



Batch Size. During the training, samples are divided into multiple batches. Samples in one batch are propagate through the network as a group. Small batch size only needs small memory, and also weights update quickly. But when batch size is too small, the weights estimation might not be accurate, as too few samples were in each batch.

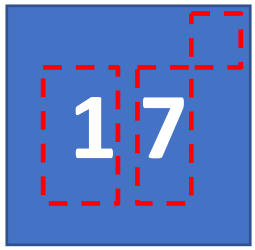
Part2. Apply the model to recognize numbers in the natural images

To recognize numbers inthe real world images, I performed the following image preprocessing to order to apply VGG model.

Deal with the noise, I have first applied a Gaussian filter to remove, and then applied a median filter to remove noise and also keep the boundary for detection.

Deal with the different scale. I have generated an image pyramid, and then search for the numbers at all levels. At each level, a sliding box strategy is applied to find the best results. The problem of this method is that depends on the distance between digits, a multidigit number might be recognized as a single digit number in the low level (big image size)

Figure3| Using *multidigit compensation* to resolve the possible conflicted results between different image levels



Level 1.
Box….
Box: detect: 1, with p=0.991
Box: detect 7, with p=0.995
Box …..
Model predicts: 7 with p=0.965
No compensation for single digit
compensated p=0.965

Level 2.
Box….
Box: detect: 17, with p=0.991
Box …..
Model predicts: 17 with p=0.961
Multidigit compensation of 0.01
compensated p=0.971

Level 3.
Box….
Model predicts: 17 with p=0.951
Multidigit compensation of 0.01
compensated p=0.971

Final prediction: The number with the highest compensated p. So it will be 17 in this example

Deal with the different number location. I applied a sliding window strategy, to search for the best numbers in the candidate regions. However, using a CNN detector across the entire image will be very computational heavy. Therefore, I find get the candidate regions by using CV2 morphological operations [8]. Then I only detect and compare all the candidate regions.

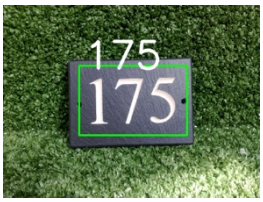
Part 3. Final results and Discussion

Link to the video

https://youtu.be/FMKfXjGTFII

Examples of the correct detection. Except the first image is taken by me in the neighborhood. All the others are from internet

Positive results



In the above 5 images, the detector performs very well and correct find all the digits. Compared with the negative images, they all have fewer pattered background.

Negative results

below are two examples of the negative examples. In fact, I found that when using the uncropped real world images, the classifier performance is significantly worse than with the SVHN data set. There are two main reasons: (1). To accelerate the detection, I detect the candidate regions first by the morphological operation. Unfortunately, that the candidate regions might not be good. Sometimes might be completely off. (2). In the training sample, we do not have negative images which completely have no numbers. So, the detector often considers certain a region with high image gradient (left image below) as a number. For example, in the left image below, the region inside the green box right does looks like “1” even to our eye. Another example is the right image below, which has overall shape similar to “4”

Possible improvement
I would like to try these three directions. (1). Generate negative training images. Those images will be only background with no numbers. (2) Add some “difficult” training samples. Such as the above 2 images. (3). I will train another CNN just to find the bounding boxes of the numbers using regression.

Refer
[1] http://ufldl.stanford.edu/housenumbers/
[2] https://pdfs.semanticscholar.org/c318/e6b00b2c9c983302713dcf36ebb42000c261.pdf
[3] https://arxiv.org/pdf/1312.6082.pdf
[4] https://www.kaggle.com/olgabelitskaya/svhn-digit-recognition
[5]https://github.com/hangyao/street_view_house_numbers/blob/master/3_preprocess_multi.ipynb
[6] http://wiki.fast.ai/index.php/Log_Loss
[7] http://cs231n.github.io/neural-networks-3/
[8] https://stackoverflow.com/questions/24385714/detect-text-region-in-image-using-opencv