Optical Character Recognition in Images using Neural Networks

Anant Agarwal (aa2387), Deekshith Belchappada (db786)

December 9, 2016

Abstract

We propose a design and implementation of Optical Character Recognition method using neural networks. We list extensions to the project and evaluation methods for the model. We also plan to experiment and measure effectiveness of different neural networks in character recognition task.

Goal

Implement an optical character recognition application for images using neural networks. Experiment and report results in the process of building the application.

Description

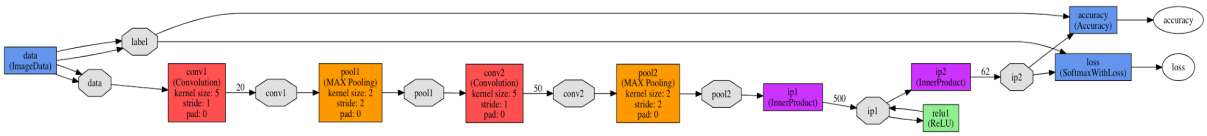
We implement character recognition task using feed forward neural network with backpropagation learning and analyse various aspects of the system. Given an image, we run segmentation to extract individual characters and detect them using the neural network built for classification task. Neural network we have built is a classifier for A-Z a-z 0-9 which is trained using manually generated data as well as the data available on internet. We have experimented with tuning various parameters related to neural network structure like the types of layers, number of iterations, training methodology, etc. Experiments section lists all the experiments with results we have seen.

Neural Network Architectures

We used the following two architectures for our experiments:

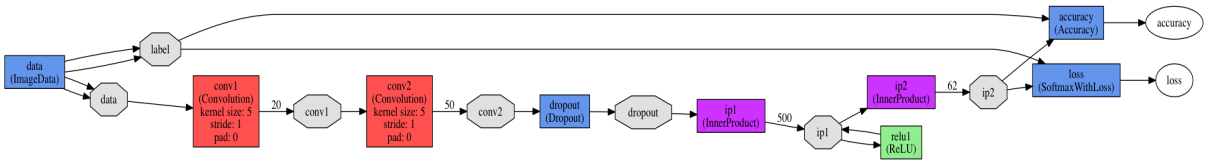
LeNet with 62 classes:

We modify the sample LeNet model architecture provided in the caffe package to correspond to our use case. Mainly we change the number of output classes to 62 (a-z A-Z 0-9), original LeNet just had 10 classes (0-9). The architecture used by us is described in the following diagram:



LeNet Modified:

We try adding a new dropout layer and removing the pooling layers from the LeNet architecture. The resultant architecture is shown in the diagram below:



We describe some of the specifics of these architectures below:

Layer Type	**LeNet**	**LeNet Modified**
Training Data Size:	5580	5580
Input layer:	scale: 0.00390625 batch_size: 64	scale: 0.00390625 batch_size: 64
Convolution Layer 1:	Kernel Size is 5 Stride is 1 num_output 20	Kernel Size is 5 Stride is 1 num_output 20
Pool Layer 1	Type: MAX Kernel_size: 2 Stride: 2	*Absent*
Convolution Layer 2	Num_output is 50 Kernel_size is 5 Stride is 1	Num_output is 50 Kernel_size is 5 Stride is 1
Pool Layer 2	Type: MAX Kernel_size: 2 Stride: 2	*Absent*
Dropout	*Absent*	Only in training Probability of dropping: 0.5
Inner Product Layer 1	Num_output: 500	Num_output: 500
ReLU Layer 1	Max(0,x)	Max(0,x)
Inner Product Layer 2	num_output: 62	num_output: 62
Accuracy Layer 1	Include Only in Test phase	Include Only in Test phase
Softmax With Loss	Only during Training phase	Only during Training phase

We have used caffe framework developed by Berkeley Vision and Learning Center and the community for implementing neural networks. We provide a brief overview of parts of Caffe which are relevant to our system in the next section.

Caffe Overview

Caffe Installation:

Caffe is an open source framework which provides APIs to build and train neural network. Following two are excellent resources for installing caffe and were used by us:
http://caffe.berkeleyvision.org/install_osx.html
http://installing-caffe-the-right-way.wikidot.com/start

Caffe Layers:

Caffe provides a predefined set of layers which can be plugged in the model and used as desired. The complete list of all the layers and the arguments they support they support can be found in <Caffe-Root>/src/caffe/proto/caffe. proto where <caffe-root> is the caffe root. We have used the following layers, a brief description of the layers are provided:

i)Image Data:
Input layer which takes a file having a list of images along with labels and the batch size, i.e. the number of images to be fetched together. Generally used while training.

ii)Input:
Input layer for the model, when it is ready to be deployed. The idea here is that we would like to pass in a set of images to the model from a separate program (e.g. passing an image to be classified from Python program to the model)

iii) Convolution:
The Conv layer is the core building block of a Convolutional Network that does most of the computational heavy lifting. It convolves the input image with a set of learnable filters, each producing one feature map in the output image.

iv) Pooling:
Its function is to progressively reduce the spatial size of the representation to reduce the amount of parameters and computation in the network, and hence to also control overfitting. It operates independently on every depth slice of the input and resizes it spatially, using the MAX operation.

iv) InnerProduct:
The InnerProduct layer (also usually referred to as the fully connected layer) treats the input as a simple vector and produces an output in the form of a single vector (with the blob's height and width set to 1).

v) ReLU:
It acts as a thresholding layer with a threshold set to 0 and performs a max(0,x) function.

vi) Accuracy:
It scores the output as the accuracy of output with respect to target, i.e. during the test iteration which happens after every few training iterations, it will generate labels for all the test images and report the accuracy.

vii) SoftmaxWithLoss:
The softmax loss layer computes the multinomial logistic loss of the softmax of its inputs. It's conceptually identical to a softmax layer followed by a multinomial logistic loss layer, but provides a more numerically stable gradient.

viii) Dropout:
The dropout layer reduces overfitting by randomly omitting each of the hidden unit from the network with some probability(0.5 in our case)

Caffe Train File:

A file containing the training data needs to be created (TrainingIndex.txt). This file is used at the time of training the model. Some of the rows from the file are shown below to serve as a sample:
Fnt/Sample007/img007-00795.png 6
Fnt/Sample029/img029-00717.png 28
Fnt/Sample043/img043-00247.png 42
First element is the image path and the second element is class label. An important thing to keep in mind is that the labels should range from 0 to <number of classes-1> (61 in our case)

Caffe Testing File:

A file containing the testing data needs to be created(TestIndex.txt). This file is used for evaluating performance at the time of training. Some of the rows from the file are shown below to serve as a sample:
Fnt/Sample007/img007-00971.png 6
Fnt/Sample011/img011-00996.png 10
Fnt/Sample001/img001-00953.png 0
Formatting is same as the train file. First element is the image path and the second element is class label. The test data and train data should be different.

Caffe Model File

This file describes the model of the neural network (generally named as Model.prototxt:). The model can be built using the various layers provided by caffe. Some of the layers that we used are described above.

Caffe Model Deploy File:

This file is same as the Model file just that we drop some layers that are required only during the training and testing step, and modify the input layer to not take the input from a file.

Caffe Solver File:

Generally named as Solver.prototxt. The solver.prototxt is a configuration file used to tell caffe how we want the network trained. This also has the reference to model.prototxt file. Parameters of the solver file are described below.

Parameters:

1. base_lr: This parameter indicates the base (beginning) learning rate of the network. The value is a real number (floating point). We experimented with a base_lr of 0.001 and 0.0001
2. lr_policy: This parameter indicates how the learning rate should change over time.
3. max_iter: This parameter indicates when the network should stop training. The value is an integer indicate which iteration should be the last. We have experimented with multiple values like 100, 1300, 10000
4. momentum: This parameter indicates how much of the previous weight will be retained in the new calculation. This value is a real fraction. We have set this to 0.9
5. weight_decay: This parameter indicates the factor of (regularization) penalization of large weights. This value is a real fraction. We have set this to 0.0005
6. Solver_mode: This parameter indicates which mode will be used in solving the network. We have set this to 'CPU' as we had chosen this option while installing caffe.
7. Snapshot: This parameter indicates how often caffe should output a model and solverstate. This value is a positive integer.
8. Snapshot_prefix: This parameter indicates how a snapshot output's model and solverstate's name should be prefixed. This value is a double quoted string.
9. Net: This parameter indicates the location of the network to be trained (path to train.prototxt). This value is a double quoted string.
10. Test_iter: This parameter indicates how many test iterations should occur per test_interval. This value is a positive integer.
11. Test_interval: This parameter indicates how often the test phase of the network will be executed.
12. Display: This parameter indicates how often caffe should output results to the screen. This value is a positive integer and specifies an iteration count.

Caffe Model Training:

Once all the files described above are ready, we are ready to train the model. The following command can be used for training the model:

caffe train -solver solver.prototxt

Caffe will store the model every few iterations (as specified in solver file)

Generation of train/test data

i. Python script for Generating Data
We looked at the most frequently used fonts for texts on internet(http://web.mit.edu/jmorzins/www/fonts.html) and chose the following fonts: "Arial", "Times New Roman", "Courier New", "Verdana", "Georgia", "Comic Sans MS", "Arial Black", "Impact", "Trebuchet MS", "Arial Narrow". We fetched all these fonts from our system and used OpenCV with Python to draw all the characters (a-z A-Z 0-9). As all the fonts can draw the same character in different size, we make sure that the all the character images generated are of equal shape and have the character in center. This results in 620 images i.e. 10 images per character. We split this data in a 90-10 fashion for training and testing.

ii. Internet
From the internet we pick "The Chars74K dataset" which is hosted by Centre For Vision, Speech and Signal Processing, University of Surrey. The data set has all the characters from a-z A-Z 0-9 and has 1016 images for each of the characters. We split each set in a 90-10 fashion and get 5580 images for training.

Segmentation

Given an image which has words, we detect words and then detect individual character in every word and write them into separate file. Once all the characters are generated as a part of segmentation, these become the input for the neural networks. Neural network performs the classification task which labels these images as per the model built during training.

We used opencv and numpy module to perform most of the segmentation task. Converted the image read using opencv libraries to gray scale so that we could perform thresholding. Applied dilation and erosion to binary image obtained after thresholding to join the gaps. Using opencv libraries, contours are found and are sorted. Parameters were set such that contours are drawn in word level (not per individual character).

Once we have word level contours in sorted order, we extract the words from the original image and draw contours within the word.This helps us in finding individual characters, we draw these characters in appropriate size and save them as separate files. These output files are passed as input to the model.

Training neural networks

We built the models described in the *Neural Networks architecture* section using caffe and trained them using the command described in *Caffe Model Training* section. The parameters used for training are already described in *Caffe Solver file Parameters*.

Using the model in Python

To integrate the segmentation part and the Neural Networks trained, we reference the model trained by Caffe from python, and then pass every character which needs to be identified to the model and make the predictions. We present the code we wrote for this purpose in a generic fashion below:

```
import caffe
import numpy as np
net = caffe.Net(<path to model_deploy.prototxt>, <path to Model.caffemodel>, caffe.TEST)
img = caffe.io.load_image(<image to be classified>)
img = img.transpose((2,0,1)) //This is to account for the different RBG conventions
img = img[None,:] //This is to say that the  batch has just one image
out = net.forward_all(data = img)
prob = net.blobs['prob'].data[0]
sorted_index = np.argsort(prob)
print(sorted_index[-1]) //This is the class label
```

Experiments and Evaluation methods

a. Shuffle/unshuffle Training Data:

In this experiment, we trained the neural network with two types of data set. Motivation behind this experiment was to understand the order in which neural network should be trained.

First scenario was to train neural network without shuffling the input data. In this case, every batch in a iteration trained all the images belonging to one class instead of training images belonging to multiple classes. For example all the images labelled A are trained first and then B and so on.

Second method was to shuffle the input data and use it for training. In this case images belonging to all the classes were trained in every batch of iteration and images were shuffled in random order.

We noticed that performance of neural network trained with no data shuffling was significantly less when compared to the neural network trained with shuffled data.

Test set size: 62 images

Train set size: 558 images.

```
Iteration 600, Testing net (#0)
I1126 19:07:16.972307 1939402752 solver.cpp:404]    Test net output #0: accuracy = 0.19625
I1126 19:07:16.972573 1939402752 solver.cpp:404]    Test net output #1: loss = 3.66814 (* 1 = 3.66814 loss)

Iteration 600, Testing net (#0)
I1123 18:44:12.278234 1939402752 solver.cpp:404]    Test net output #0: accuracy = 0.6517
I1123 18:44:12.279240 1939402752 solver.cpp:404]    Test net output #1: loss = 0.65518 (* 1 = 0.65518 loss)
```

Conclusion: For all the later experiments, we trained the network with shuffled data.

b. Training set size

Motivation behind this experiment was to understand the impact of training data set size. We tried with various train data set size. First attempt was to try with 558 images covering all 62 classes and train the network. Second attempt was with 5580 images and finally third attempt was with 55800 images. Training our model with 55800 images took lot of time as we were using caffe in CPU mode and we decided to do further experiments with smaller data set size.

Model trained with 558 images had an accuracy of 65.51%. Model trained with 5580 images achieved an accuracy of 78.15%.

Conclusion: Model becomes better as we increase the training data size.

c. Number of Iterations:

In this experiment, we trained our neural network with various iteration parameters and compared the results. As we increased the number of iteration, accuracy of neural network increased. Following paragraph captures the accuracy of neural network which was trained with 5580 images.

18:51:25.492118 1953816576 solver.cpp:337] Iteration **1000**, Testing net (#0)
18:51:26.672399 1953816576 solver.cpp:404] Test net output #0: accuracy = **0.6075**

20:13:45.331542 1953816576 sgd_solver.cpp:106] Iteration **2000**, lr = 7.23368e-06
20:15:33.165902 1953816576 solver.cpp:337] Iteration 5500, Testing net (#0)
20:15:33.734064 1953816576 solver.cpp:404] Test net output #0: accuracy = **0.6250**

21:39:02.101899 1953816576 solver.cpp:317] Iteration **3000**, loss = 0.275028
21:39:02.101951 1953816576 solver.cpp:337] Iteration 10000, Testing net (#0)
21:39:03.187620 1953816576 solver.cpp:404] Test net output #0: accuracy = **0.7815**

Conclusion: Training the model with higher number of iterations will help in improving accuracy.

d. Experiments with layers:

We experimented with two different neural network layers as explained in *Neural Networks Architecture* section. Model trained as per basic lenet structure showed an accuracy of 78% when trained with 5580 images. On the other hand, model trained with modified lenet structure showed an accuracy of 86% when trained with 5580 images.

Conclusion: Removing pooling layers and adding a dropout layer helped in increasing performance.

e. **Experiments with learning rate**:

We tried using different learning rates, we started with the rate 0.01 and went down to 0.0001. With a higher learning rate the model didn't converge.

Conclusion: Keep reducing the learning rate till the model starts to move towards minima.

Future work

1. Generate training data using segmentation, label them manually and train neural network on that data. This will show better results even on test set generated by segmentation, and hence better performance on recognizing characters from images.

2. We have trained our model with input data size of 5580 images because we were using CPU mode. Installing caffe in GPU mode and training neural network with large input size will give even better performance.

3. A deeper neural network can be created by adding more convolutional layer and ReLu layers, this should result in better performance.

4. Extend this application to work on videos. Take snapshots of videos per few frames, and then apply the character recognition on these images to extract words written in videos.

5. Implement an application for indexing videos based on the text that appears in videos

Conclusion

We have trained our network with images created using python script and images readily available on internet and not trained with images which are created by segmentation of images having characters. Generating input data using segmentation, labelling them manually and training neural network on that data set will show better results even on test set generated by segmentation. Additionally, training on a bigger dataset with more number of layers should help in developing a production level application.

References

1. The Chars74k Dataset: http://www.ee.surrey.ac.uk/CVSSP/demos/chars74k/
2. Stanford cs231n - Convolutional Neural Networks for Visual Recognition: http://cs231n.github.io/convolutional-networks/
3. Caffe: http://caffe.berkeleyvision.org/
4. Caffe Installation: http://installing-caffe-the-right-way.wikidot.com/start
5. Segmentation Ideas: http://rnd.azoft.com/applying-ocr-technology-receipt-recognition/
6. Caffe with Python Tutorial: https://prateekvjoshi.com/2016/02/02/deep-learning-with-caffe-in-python-part-i-defining-a-layer/
7. Most frequent fonts: http://web.mit.edu/jmorzins/www/fonts.html