Spatially-sparse convolutional neural networks

Calin Botoroaga(C.T.Botoroaga@student.tudelft.nl, 5407869)
Yao Ma (Y.MA-11@student.tudelft.nl, 5222982)

Abstract

In this post, we would like to mark the following contributions as our criteria.

- New code variant:
  We followed the arrangement of the convolutional and pooling layers mentioned in the paper to build a customized DeepCNet.
- New code variant:
  This is shown in the way we deal with the sparse input. The provided codes are highly encapsulated which are hard for readers to identify how the input spatial size is transformed. We make it explicitly aligned with the one mentioned in the paper that the advantage of sparsity is related to the spatial padding for convolutional networks.
- New code variant:
  A simple CNN was implemented as a baseline for comparison with the more advanced architectures
- Reproduced:
  An existing VGG architecture provided by the author was evaluated

Introduction

Convolutional neural networks perform well on problems such as handwriting recognition and image classification. However, the performance of the networks is often limited by budget and time constraints, particularly when trying to train deep networks. The use of spatially-sparse input is a technique to speed up the computation, which also provides one with greater freedom when it comes to preparing the input data.

A more concrete example of showing the advantage of sparse input is shown in Figure 1 from the original paper[^1]. For general input, slow pooling is relatively computationally expensive as the spatial size of the hidden layers reduces more slowly. When the input array is sparse, this is offset by the fact that sparsity is preserved in the early hidden layers. Only the values of the hidden variables where they differ from their ground state are calculated. To illustrate the efficiency, we employ deep neural networks on both online character recognition and offline image recognition.

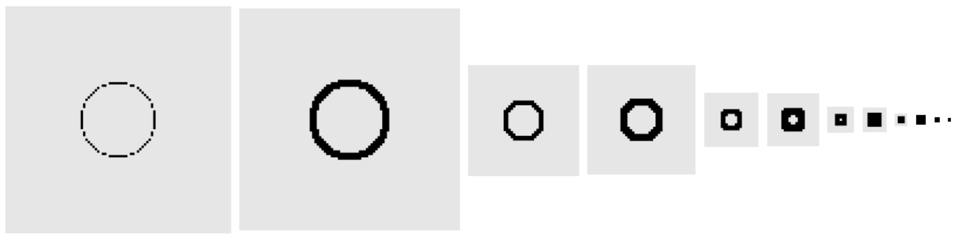


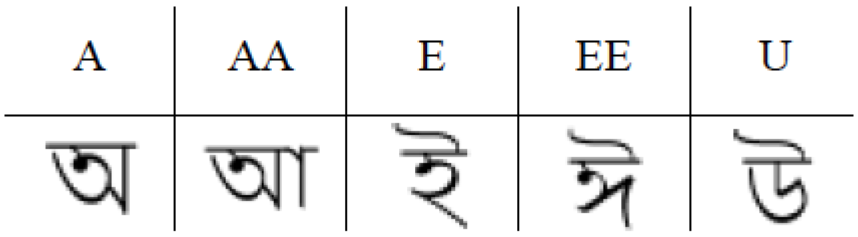
Figure 1[^1]: How the active spatial locations(black) change through the layers. Sparsity manifests itself in the early stages where most of the spatial locations are in their ground state (gray).

Experimental Setup

Datasets

Dataset of Online Handwritten Assamese Characters

The Assamese Characters dataset contains 183 classes which consist of Assamese numerals and alphabetic characters. There are a total of 8235 samples in the dataset, each class being represented by 45 samples. The digits are handwritten online, meaning the motion of the pen is also recorded such that the points that make up the different letters appear in the order in which they were written. The data is given in the form of a list of x,y coordinates that represent the points where the pen has touched the input tablet. The points are on a high resolution 4392 * 4868 grid. This means that the Assamese dataset contains very sparse data which is ideal for the architecture proposed in the paper.
A few examples of Assamese Characters and their labels can be viewed in Figure 2.



A	AA	E	EE	U
অ	আ	ই	ঈ	উ

Figure 2: Some Assamese Characters

All the actual samples of the dataset are formatted as follows: There is a header that gives the label of the character, the number of pen strokes that the particular sample contains. Individual strokes are delimited by lines with “PEN_DOWN” and “PEN_UP”. An example of such a file is given below:

```
CHARACTER_NAME: NAA
STROKE_COUNT: 1
 X          Y   STYLUS_STATE STROKE
```

```
PEN_DOWN
2170        2223        1        1
2143        2249        1        1
2143        2275        1        1
2143        2302        1        1
2117        2302        1        1
2117        2328        1        1
2090        2328        1        1
2090        2355        1        1
.
.
.
1852        1402        1        1
PEN_UP                  0
END_CHARACTER: NAA
```

For this dataset the data was loaded in 2 different ways. This is because we used it using both a sparse VGG implementation provided by the author as well as a dense simple CNN in order to better understand the differences between the 2 approaches. For the VGG the data is given to the network directly in the provided sparse format. For our simple dense CNN we created a custom pytorch dataset. The dataset class takes the sparse input from the files and converts them to dense tensors. Since the sparse grid is 4392 * 4868 this means that the resulting tensor will be more than 21 million data points in size making it very slow to train a network. Because of this, we also apply subsampling and we rescale the image to 48 * 48. The result is presented in Figure 3.

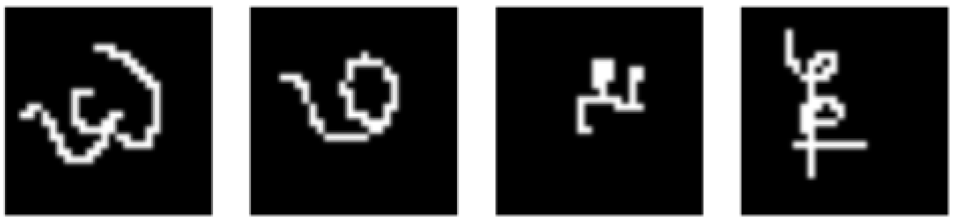


Figure 3: The Rescaled Dense Assamese Characters

As it can be observed from the figure, most of the details are well preserved after the transformation.

MNIST

MNIST is a data set composed of handwritten digits, containing 60,000 training and 10,000 test examples, and each example represents a 28x28 digital image. The author extended the training set only by translations with shifts up to +/-2 pixels, so we also adopted this method to do the transformation.

This paper originally used two DeepCNet network architectures, as shown in Table 1. The first network contains 6 convolutional layers and 5 max pooling Layers. In the second network, different dropout levels for each convolutional layer are additionally added, which are 0; 0; 0; 0.5; 0.5 ; 0.5; 0.5. The number of filters and filter size of each layer are set up according to the paper. The hidden layers all use the ReLU activation function, and the output layer uses the Softmax activation function. For the batch size and epochs, we set 256 and 80 respectively for training.

The following code shows the process of using PyTorch to obtain and augment the MNIST dataset in each training epoch.

```
from torchvision.datasets import MNIST
BATCH_SIZE = 256

# Download the MNIST dataset
train_data = MNIST(root = './', train=True, download=True, transform =
torchvision.transforms.Compose([torchvision.transforms.ToTensor()]))
test_data = MNIST(root = './', train=False, download=True, transform =
torchvision.transforms.Compose([torchvision.transforms.ToTensor()]))

# Data Loader for easy mini-batch
train_loader = Data.DataLoader(dataset=train_data,
batch_size=BATCH_SIZE, shuffle=True)
test_loader = Data.DataLoader(dataset=test_data, batch_size=BATCH_SIZE)

for i, (x_batch, y_batch) in enumerate(train_loader):

       # Set to same device
       x_batch, y_batch = x_batch.to(device), y_batch.to(device)
       x_batch = torch.cat([x_batch, torch.roll(x_batch,2,2),
torch.roll(x_batch,-2,2)]) # augment the dataset by shifts of up and
down to 2 pixels
       y_batch = torch.cat([y_batch, y_batch, y_batch])

```

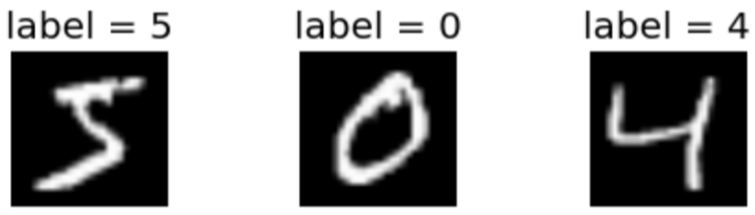


Figure 4: MNIST sample data[^2]

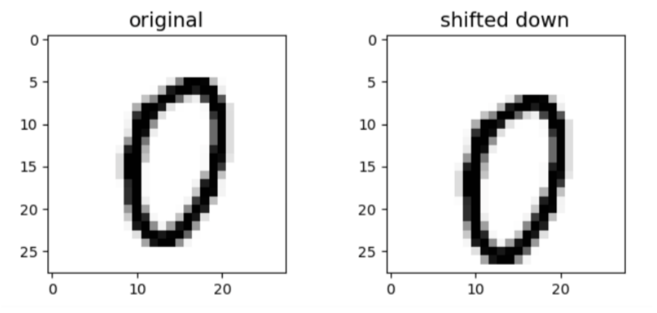


Figure 5: Augmented image by shifts of down to 2 pixels[^2]

Architecture

As in the original paper, we maintained the following networks. For the online dataset, one dense CNN and one sparse VGG are implemented for comparison. For the offline dataset, we have two DeepCNet networks with the same depth but different dropout levels, one is DeepCNet(5,10), the other one is DeepCNet(5,60). The following describes the architecture of each network.

Dense CNN

The dense CNN implemented is a simple sequential network with 3 convolutional layers, relu activation function and max pooling layers between the convolutional layers. The final layer is a fully connected one. Again, we use Adam optimizer with a learning rate of 0.001. We use this architecture for both the MNIST dataset and the dense version of the Assamese dataset.

```
class Net(nn.Module):
   def __init__(self, in_channels, hidden_channels,
out_features,output_resize):
       super(Net, self).__init__()
       self.net = nn.Sequential(
       nn.Conv2d(in_channels,
hidden_channels[0],kernel_size=3,padding=1),
       nn.ReLU(),
       nn.MaxPool2d(2),
       ...
       )
       self.fn =
nn.Linear(output_resize*output_resize*hidden_channels[2],
out_features)

   def forward(self, x):
       x = self.net(x)
       x = x.view(x.size(0), -1)
       x = self.fn(x)
       return x
```

Sparse VGG network

The author provides an example of a sparse VGG network that we were able to run on the Assamese dataset. Unfortunately VGG isn't a network presented in the paper, however we have found it useful to run this example as a way of testing the claims of the paper. This is because while the network might not be identical, it is similar to the proposed architectures. Moreover it is a good example of how the idea of sparsity can be applied. VGG is a very deep convolutional network and because of this it is usually incredibly slow to train. However, by making use of sparsity, the network is actually sped up significantly.
The actual architecture of the network is depicted in figure 6.

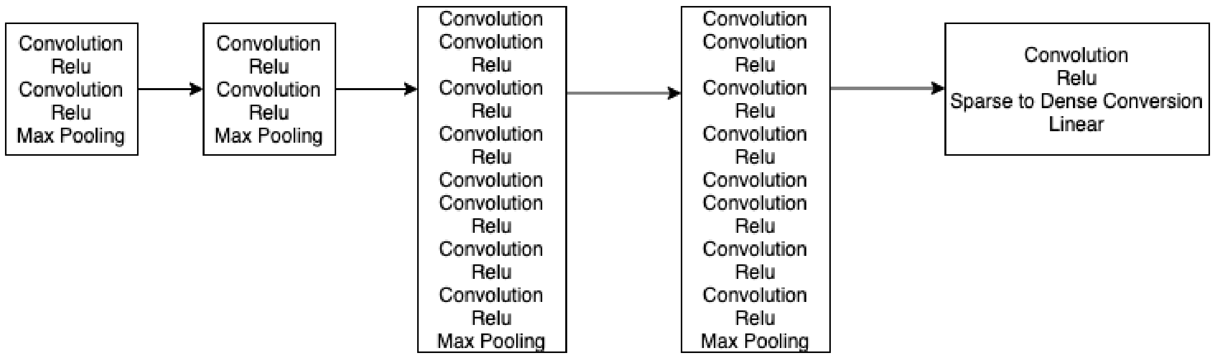


Figure 6: Diagram of the VGG architecture provided by the author

DeepCNet(l, k)

For the MNIST dataset, we used the Adam algorithm to calculate the gradient descent. The initial learning rate is 0.001. This value is stable in our pre-training, so we keep using this value in the formal training. In the network class below, you can find the code for the specific DeepCNet(5,10) architecture without dropout. The DeepCNet requires the input layer has input spatial size N*N, where N = 3*2^l. For the first convolutional layer, the filter size is set to 3*3 while the following convolutional layers have the filter size of 2*2.
```
class Net(nn.Module):
   """
   Args:
in_channels: number of features of the input image
hidden_channels: number of hidden features
out_features: number of features in output layer
   """


   def __init__(self, in_channels, hidden_channels, out_features):
       super(Net, self).__init__()

       self.conv1 = nn.Sequential(         # input shape
           nn.Conv2d(in_channels=in_channels,
out_channels=hidden_channels[0], kernel_size=3, stride=1, padding=1),
           nn.ReLU(),
nn.MaxPool2d(kernel_size=2) ) # choose max value in 2x2

     ...
       self.conv6 = nn.Sequential(
           nn.Conv2d(hidden_channels[4],hidden_channels[5],
kernel_size=2, stride=1, padding=1),
           nn.ReLU(),
       )
       self.fc = nn.Linear(4*4*hidden_channels[5], out_features)

   def forward(self, x):
       x = self.conv1(x)
       x = self.conv2(x)
       x = self.conv3(x)
       x = self.conv4(x)
       x = self.conv5(x)
       x = self.conv6(x)
       x = x.view(x.size(0), -1)
       x = self.fc(x)
       return x
```

During the training loop, one essential implementation is making the input spatial space size as sparse. We padded zero on the original MNIST image size which is 28*28, thus a larger input size 96*96 is made as the sparse input. This smart and easy implementation avoids the burden of using complex definitions from the author's Github repository. Although the number of epochs can be modified, the model becomes stable in about 80 epochs. In order to visualize our model, we drew the curve of the accuracy on the training set and the test set respectively.

```
for epoch in range(epochs):
    # Network in training mode and to device
    net.train()
    net.to(device)
    # Training loop
    for i, (x_batch, y_batch) in enumerate(train_loader):
        # Set to same device
        x_batch, y_batch = x_batch.to(device), y_batch.to(device)
        x_batch = nn.functional.pad(x_batch, (34,34,34,34,0,0,0,0,),
'constant', 0)
```

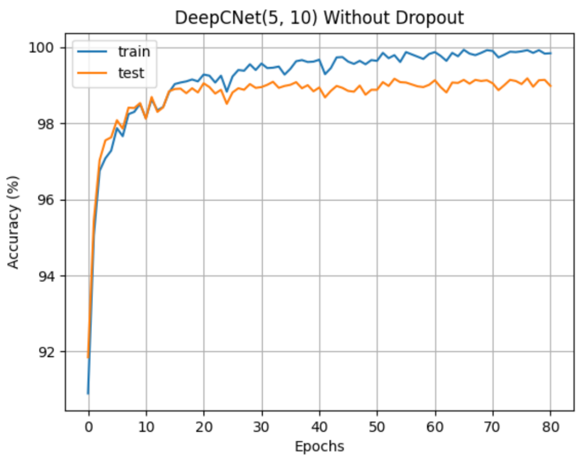
Results and Evaluation

In the following section, we discuss the results of our networks. Since some hyperparameter setting details were not covered in the original paper, we didn't expect to get the same exact values. We tried to maintain our architecture as similar as discussed in the paper, so we believed that it would be somewhat valuable as the output.

MNIST

DeepCNet

The following are accuracy curves for our MNIST network against training steps. For the simpler network, the test error after 80 epochs has already achieved 0.86% which is pretty close to the test error 0.58% from the paper. For the network with dropout, the computation is costly so we only observe the test error 0.67% within 20 epochs. Better results could be obtained by increasing the number of epochs since the curves still have the tendency to go up.



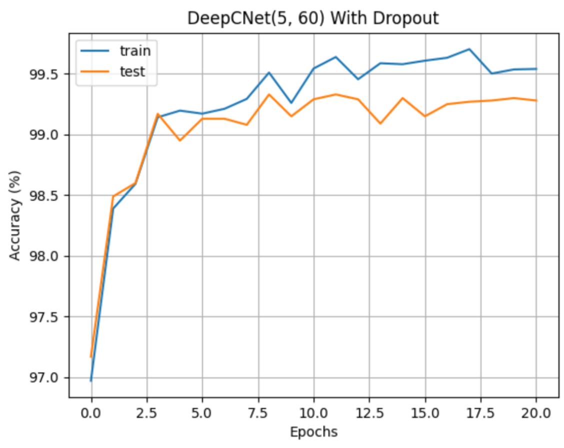


Figure 76: Accuracy on DeepCNet(5,10) without dropout and DeepCNet(5, 60) with dropout

CNN

For the MNIST dataset we have also used our simple CNN for comparison. The results are shown in figure 8. As it can be seen from the figure the accuracy for this simple network plateaus at 98%. After that only the accuracy on the training set improves so the network is overfitting after 15 epochs. This is expected as the network size is relatively small.

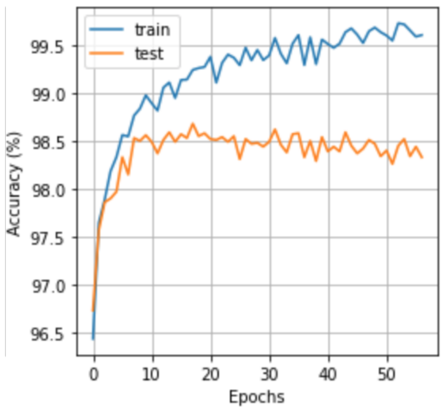


Figure 8: Accuracy for the CNN on the MNIST dataset

Dataset of Online Handwritten Assamese Characters

CNN

This network is relatively small and therefore it has a hard time predicting the 183 different classes of characters. The network peaks around 60% accuracy so we believe these results have limited use. For future work, the network size has to be increased in order to cope with the large number of classes.

VGG

The VGG network accuracy measures can be viewed in Figure 10. The accuracy is around 98.5% after 20 epochs. It is notable however that it is very quick to train. It takes roughly 3 times the amount of time needed to train our simple CNN per epoch. Taking into account the fact that the VGG is a lot deeper, this does show that there are large speed gains to be had when the input is sparse.

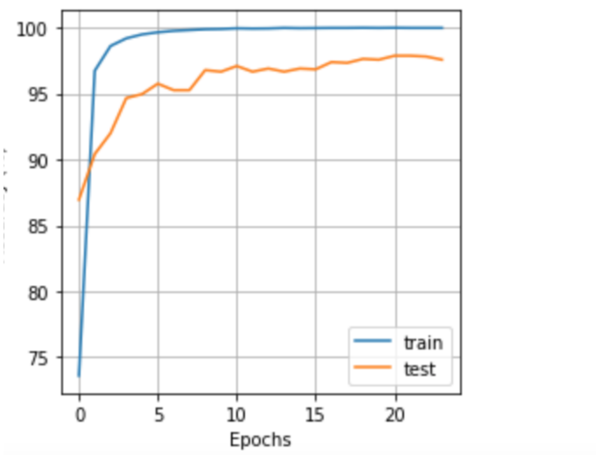


Figure 10: Accuracy for VGG

Conclusion

In this attempt to reproduce the paper “Spatially-Sparse convolutional neural networks” a simple CNN was implemented as a baseline, a VGG style network provided by the author was evaluated, a dense DeepCNet was implemented from scratch. A small CNN has been found to perform very well on the MNIST dataset, however this did not translate to a good performance on the Assamese dataset.
While the author claims a test error of 1.7% in the paper, we have found that the provided VGG has an error of only 1.5%. This might be due to the fact that the VGG implementation improves over the author’s previous DeepCNet implementation which might be the reason why that architecture is no longer provided.

We showed that the sparse representation of offline datasets has allowed us to improve the performance on the normal CNN more than 0.5%. Although our offline results from DeepCNet are a bit lower than those in the paper, we believe it is related to the number of epochs, accidental training errors and other statistical factors. We can also see more flexible transformation choices could be considered if the input is placed in a substantially larger grid.

For future work, the network structures discussed in this blog could be extended to higher-dimensional convolutional layers, which would be quite useful for fast detection and analysis of static objects.

Reference

[^1]Graham, Benjamin. "Spatially-sparse convolutional neural networks." arXiv preprint arXiv:1409.6070 (2014).

[^2]Medium. 2021. Improving accuracy on MNIST using Data Augmentation. [online] Available at: <https://towardsdatascience.com/improving-accuracy-on-mnist-using-data-augmentation-b5c38eb5a903> [Accessed 16 April 2021].
[^3]U. Baruah, S. M. Hazarika, "A Dataset of Online Handwritten Assamese Characters", Journal of Information Processing Systems, vol. 11, no. 3, pp. 325-341, 2015.