OUR REPORT TITLE*

ZICHEN CHAO†, SHENLONG GU†, KAN ZHU†, AND YIBO ZHU‡

Abstract. In this paper, we realize the result of the paper ReNet: A Recurrent Neural Network Based Alternative to Convolutional Networks [8] using Python's Deeplearning Library *Theano*.

Key words. Recurrent Neural Networks, Convolutional Neural Networks, Theano

1. Introduction.

1.1. Motivation. Convolution Neural Networks(CNN) have proven its power in object recognition and image classification problems. Previous works of Recurrent Neural Networks(RNN) also shows great performance in handwriting recognition/generation and speech recognition. Rather than the multi-dimensional RNN inspired before, one could use a uni-dimensional RNN, which makes each output activation of the layer be computed with respect to the whole input image, rather than the local extracted activation computed in CNN. Also, comparing to multi-dimensional RNN, the number of RNNs at each layer scales could be linear with respect to the number of dimensions d of the input image ($2d$), while multi-dimensional RNN requires the exponential number of RNNs at each layer (2^d). Furthermore, parallelizability could be easier, as each RNN is dependent only along a horizontal or vertical sequence of patches.

1.2. Related Work. The idea of using RNN on offline handwriting recognition is from Graves and Schmidhuber's paper[5]. They use Multi-dimensional RNN (MDRNN), which avoids any alphabet specific preprocessing and thus could be applicable for any language, with the help of connectionist temporal classification algorithm. The challenging part of offline problem comparing to online problem is that the input is no longer one dimensional, hence if the image shifts vertically by one pixel, it would be completely different. Other related works aiming at classification of images is summarized in a github website [1].

1.3. Organization. The rest of the paper is organized as follows. We review the model architecture of ReNet in §2. Then we propose our implementation of ReNet in §3. In §4 we recreate the result of ReNet. Lastly, §5 concludes the paper and gives direction for future work.

2. Background. The ReNet paper proposed a deep neural network architecture for object recognition based on recurrent neural networks(RNN). The proposed network, called ReNet, replaces the ubiquitous convolution+pooling layer of the deep convolutional neural network with four recurrent neural networks that sweep horizontally and vertically in both directions across the image.

First, we sweep the image vertically with two RNNs, with one RNN working in a bottom-up direction and the other working in a topdown direction. Each RNN takes as an input one (flattened) patch at a time and updates its hidden state, working

*This report was conducted as part of the Columbia ECBM E6040 final project.

†Department of Computer Science, Columbia University in the City of New York.

‡Department of Industrial Engineering and Operations Research, Columbia University in the City of New York.

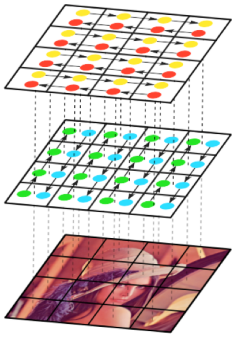
along each column j of the split input image X.

After this vertical, bidirectional sweep, we concatenate the intermediate hidden states (forward and backward) at each location of patch to get a composite feature map. Each state is now the activation of a feature detector at the corresponding location with respect to all the patches in the j-th column of the original input. Next we sweep over the obtained feature map horizontally with two RNN. In a similar manner as the vertical sweep, these RNNs work along each row resulting in the output feature map. Now, each vector represents the features of the original image patch in the context of the whole image.

The functions are as follows:

$$v^F_{i,j} = f_{VFWD}(z^F_{i,j-1}, p_{i,j}) \text{ for } j = 1, \ldots, J$$
$$v^R_{i,j} = f_{VREV}(z^R_{i,j+1}, p_{i,j}) \text{ for } j = J, \ldots, 1$$

The schema is as follows:



They evaluate the proposed ReNet on three widely-used benchmark datasets; MNIST, CIFAR-10 and SVHN. The result suggests that ReNet is a viable alternative to the deep convolutional neural network, and that further investigation is needed.

3. Implementation.

3.1. Data Preprocessing. It is well-known that augmenting training data often leads to better generalization. We employed a shifting augmenting strategy. In the case of shifting, we either shifted the image by 2 pixels to the left (25% chance), 2 pixels to the right (25% chance) or left it as it was. After this first processing, we further either shifted it by 2 pixels to the top (25% chance), 2 pixels to the bottom (25% chance) or left it as it was. This two-step procedure makes the model more robust to slight shifting of an object in the image. The shifting was done without padding the borders of the image, preserving the original size but dropping the pixels which are shifted out of the input while shifting in zeros. The choice of whether to apply these augmentation procedures on each dataset was chosen on a per-case basis in order to maximize validation performance.

Here is snapshots on parts of our data augmentation code:

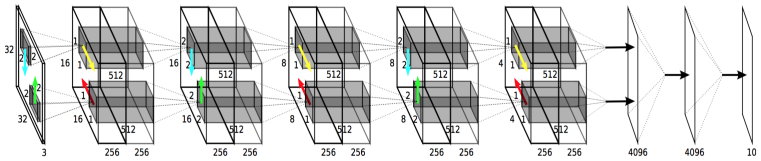
```
    def shift(data, dim):
        dimension = data.shape[0]
        edge = int(math.sqrt(dimension / 3))
        dirx = [0, 0, 2, -2]
        diry = [-2, 2, 0, 0]
        res = numpy.zeros((dimension,), data.dtype)
        for j in range(0, edge):
            for k in range(0, edge):
                newx = j + dirx[dim]
                newy = k + diry[dim]
                if newx < 0 or newx >= edge or newy < 0 or newy >= edge:
                    continue
                idx = j * edge + k
                newidx = newx * edge + newy
                res[3 * newidx] = data[3 * idx]
                res[3 * newidx + 1] = data[3 * idx + 1]
                res[3 * newidx + 2] = data[3 * idx + 2]
        return res
    # if data augmentation then do it
    if aug:
        n, dimension = train_set[0].shape
        # 25% l 25% r
        for i in range(0, n):
            r = random.uniform(0,1) * 4
            if r < 1:
                train_set[0][i] = shift(train_set[0][i], 0)
            elif r < 2:
                train_set[0][i] = shift(train_set[0][i], 1)
        # 25% u 25% d
        for i in range(0, n):
            r = random.uniform(0,1) * 4
            if r < 1:
                train_set[0][i] = shift(train_set[0][i], 2)
            elif r < 2:
                train_set[0][i] = shift(train_set[0][i], 3)
```

Meanwhile, we also pre-processed image data by scaling its original pixel to be mean zero.

3.2. Adaptive Learning Rate. We applied adaptive learning rate algorithm to achieve better test accuracy. Based on what have learned in class, we proposed a very straightforward algorithm to set the adaptive learning rate. The basic idea is if we find the discrepancy between validation errors of two consecutive steps are very small, we think the learning rate maybe too large which cannot approximate optimal value well. So we reduce the learning rate at this time. And we should speed up the learning rate reduction as training process goes further. Because the more we are closed to the optimal point, the smaller step we need to choose and the more cautious we should be at learning rate. Therefore, weve tried several methods of learning rate reduction. Experiments told us our choice works well. The following code shows how we changed the learning rate:

```
                if adapt and (best_validation_loss-this_validation_loss)*100 < 0.2:
                    l_r.set_value(learning_rate/(1+d_lr*0.1))
                    d_lr += 1
```

3.3. Neural Network Implementation. We implement our RNN demonstrated in the above figure. First, we input our image data to ReNet layer, then to the MLP Layer and finally to the 10-mode output layer.



For a ReNet layer, first, we sweep the image vertically with two RNNs, with one RNN working in a bottom-up direction and the other working in a topdown direction. Each RNN takes as an input one (flattened) patch at a time and updates its hidden state, working along each column j of the split input image. After this vertical, bidirectional sweep, we concatenate the intermediate hidden states (forward and backward) at each location of patch to get a composite feature map. Each state is now the activation of a feature detector at the corresponding location with respect to all the patches in the j-th column of the original input. Next we sweep over the obtained feature map horizontally with two RNN. In a similar manner as the vertical sweep, these RNNs work along each row resulting in the output feature map. Now, each vector represents the features of the original image patch in the context of the whole image. Moreover, the hidden state of each RNN that we used is Gated recurrent units. The hidden state of the GRU at time t is computed by [4]:

$$
\begin{aligned}
h_t &= (1 - \mu_t) \odot h_{t-1} + \mu_t \odot \bar{h}_t \\
\bar{h}_t &= \tanh(W x_t + U(r_t \odot h_{t-1}) + b) \\
[u_t; r_t] &= \sigma(W_g x_t + U_g h_{t-1} + b_g)
\end{aligned}
$$

4. Result. We've tested the model under multiple combination of hyper-parameters for the CIFAR-10 dataset, and here is what we get:

CIFAR-10

#Schema	d	hu	h	$batch$	aug	$adapt$	Test error
with augmentation	80	500	1	200	T	T	29.41%
without augmentation	80	500	1	200	F	T	28.21%
less hidden state of renet	40	500	1	200	F	T	28.67%
less hidden unit of mlp	80	200	1	200	F	T	29.76%
more hidden layer of mlp	80	200	4	200	F	T	31.42%
No adapt learning	80	500	1	200	F	F	29.97%

We set d as the number of hidden units in each ReNet layer, hd as the number of hidden units in each layer, h as the number of hidden layer, $batch$ as the batch size, aug as whether we use data augmentation preprocessing, $adapt$ as whether we use adaptive learning rate.

We've also tested the model for MNIST dataset, we managed to get 1.53% test error given the parameter set as $\{d, hu, h, batch, aug, adapt\} = \{40, 250, 1, 200, F, F\}$.

The test results are summarized in the above table. The first two columns show that with data augmentation, we do not gain a test error decrease, instead, the test error increased slightly. The 2nd and the 3rd column shows that we reduce the test error by 0.4% with higher number of units in ReNet layer. The 2nd and the 4th

column show that we reduce the test error by 1% with higher number of hidden units. The 4th and 5th column shows that we reduce the test error by 2% with less number of hidden units in MLP. The 2nd column and 6th column show that we reduce the test error by 2% with adapted learning rate.

This is part of our main function implementing the ReNet model:

```
class ReNet(object):
    def __init__(self, input, batch_size, w, h, c, wp, hp, d, unit_option="gru"):
        '''
        input: last layer output, or the initial image
        w: input width
        h: input height
        batch_size: the number of the input sample
        c: input channel num
        wp: width patch length
        hp: height patch length
        d: hidden unit dimension
        After the four direction rnn the output will be size [batch_size, (w / wp), (h / hp), 2 * d]
        '''
        # first get left to right right to left hidden expression then stack
        if unit_option == "gru":
            l_to_r = GruReNetDir(input, batch_size, w, h, c, wp, hp, d, 2)
            r_to_l = GruReNetDir(input, batch_size, w, h, c, wp, hp, d, 3)
        elif unit_option == "lstm":
            l_to_r = LstmReNetDir(input, batch_size, w, h, c, wp, hp, d, 2)
            r_to_l = LstmReNetDir(input, batch_size, w, h, c, wp, hp, d, 3)
        # stack together
        output1 = T.concatenate([l_to_r.output, r_to_l.output], axis=3)
        # up to down and down to up
        if unit_option == "gru":
            u_to_d = GruReNetDir(output1, batch_size, w / wp, h / hp, 2 * d, 1, 1, d, 0)
            d_to_u = GruReNetDir(output1, batch_size, w / wp, h / hp, 2 * d, 1, 1, d, 1)
        elif unit_option == "lstm":
            u_to_d = LstmReNetDir(output1, batch_size, w / wp, h / hp, 2 * d, 1, 1, d, 0)
            d_to_u = LstmReNetDir(output1, batch_size, w / wp, h / hp, 2 * d, 1, 1, d, 1)
        # get the output
        self.output = T.concatenate([u_to_d.output, d_to_u.output], axis=3)
        self.test = theano.function([input], self.output)
        # get the paramters
        self.params = l_to_r.params + r_to_l.params + u_to_d.params + d_to_u.params
```

5. Discussion. We successfully implemented the ReNet model. The result is perfect comparing to some previous works. However, we need to be fully aware of some problems we are facing:

First, computing power limits. We've tested our model on AWS GPU instance with 1× NVIDIA GRID GPU (Kepler GK104) and 8× hardware hyperthreads from an Intel Xeon E5-2670. This is the best configuration that we could afford and access now. However, this also limits the training performance. Hence the hyper-parameters aren't fully tested. One could observe that our best performance happens when we choose larger parameters for hidden units and dimension of ReNet layers.

Second, time limit. Due to the fact that this project is due within two weeks, we don't have sufficient time to fully test the performance of the ReNet model.

Thirdly, budge limit. We have spent more than 300 dollars on this projects. Although we fully devote ourselves into this project, these are really serious obstacles spotted on our way to pursue perfectness.

5.1. Future Work. First, we evaluated the proposed ReNet only quantitatively. However, the accuracies on the test sets do not reveal what kind of image structures the ReNet has captured in order to perform object recognition. Further investigation along the line of regularization of neural networks using dropconnect is necessary, as well as exploring ensembles which combine RNNs and CNNs for bagged prediction.[2] [3][6][7][9]

Second, we can use many parallelization tricks which are widely used for training CNN such as parallelizing fully-connected layers, having separate sets of kernels/features in different processors and exploiting data parallelism.

Third, this mechanism can be applied to multidimensional grid other than image. We can applied the algorithm to 3D real object detection.

Contribution. Zichen Chao analyzed the algorithm, did the coding part, debugged the program, optimized the model performance. Shenlong Gu designed the whole system of ReNet, analyzed the algorithm, optimized the performance, and did the coding part. Kan Zhu analyzed the algorithm, optimized the performance, wrote the report and refined the model. Yibo Zhu analyzed the model, raised algorithmic suggestions, debugged the codes and wrote the report.

REFERENCES

[1] *What is the class of this image?* Accessed: 2016-04-28.

[2] Sean Bell, C. Lawrence Zitnick, Kavita Bala, and Ross B. Girshick, *Inside-outside net: Detecting objects in context with skip pooling and recurrent neural networks*, CoRR, abs/1512.04143 (2015).

[3] Liang-Chieh Chen, Jonathan T. Barron, George Papandreou, Kevin Murphy, and Alan L. Yuille, *Semantic image segmentation with task-specific edge detection using cnns and a discriminatively trained domain transform*, CoRR, abs/1511.03328 (2015).

[4] Kyunghyun Cho, Bart van Merrienboer, Çaglar Gülçehre, Fethi Bougares, Holger Schwenk, and Yoshua Bengio, *Learning phrase representations using RNN encoder-decoder for statistical machine translation*, CoRR, abs/1406.1078 (2014).

[5] Alex Graves and Juergen Schmidhuber, *Offline handwriting recognition with multidimensional recurrent neural networks*, in Advances in Neural Information Processing Systems 21, D. Koller, D. Schuurmans, Y. Bengio, and L. Bottou, eds., Curran Associates, Inc., 2009, pp. 545–552.

[6] Pan He, Weilin Huang, Yu Qiao, Chen Change Loy, and Xiaoou Tang, *Reading scene text in deep convolutional sequences*, CoRR, abs/1506.04395 (2015).

[7] Nal Kalchbrenner, Ivo Danihelka, and Alex Graves, *Grid long short-term memory*, CoRR, abs/1507.01526 (2015).

[8] Francesco Visin, Kyle Kastner, Kyunghyun Cho, Matteo Matteucci, Aaron C. Courville, and Yoshua Bengio, *Renet: A recurrent neural network based alternative to convolutional networks*, CoRR, abs/1505.00393 (2015).

[9] Zhen Zuo, Bing Shuai, Gang Wang, Xiao Liu, Xingxing Wang, and Bing Wang, *Learning contextual dependencies with convolutional hierarchical recurrent neural networks*, CoRR, abs/1509.03877 (2015).