1 Introduction

This report summarized some experiments about asymmetric backpropagation. The backpropagation algorithm is crucial to neural network training, and it's based on the chain rule of derivatives. However, when backpropagation is viewed in a way of neural signal feedback, it's not biological plausible since the backward signal in real neural system is through a independent path. This limitation is due to the fact that neural cells are all one way cells. Asymmetric backpropagation do not use the information of weights in the forward propagation, which leads to a new form of neural network training.

In the neural network implemeted by "graph based" auto differentiation, typical dense layers is:

$h_{n+1} = f(h_n W)$

where h_n is the preception state value of layer n and W is the connection weight from layer n to layer n+1. f is the nonlinearity, possibly RELU. In real implemention, h is a matrix with row for batch size and p column for features at this layer, W has p row for feature in the layer n and p' column for feature in layer $n+1$. In the original backpropagation, let L be our final loss, then according to the chain rule:

$\frac{\partial L}{\partial W} = \frac{\partial L}{\partial h_{n+1}} \frac{\partial h_{n+1}}{\partial h_n W} \frac{\partial h_n W}{\partial W} = h_n^T (\frac{\partial L}{\partial h_{n+1}} * f'(h_n W)) = dot(h_n^T, G)$

$\frac{\partial L}{\partial h_n} = \frac{\partial L}{\partial h_{n+1}} \frac{\partial h_{n+1}}{\partial h_n W} \frac{\partial h_n W}{\partial h_n} = (\frac{\partial L}{\partial h_{n+1}} * f'(h_n W)) W^T = dot(G, W^T)$

Here G denote value matrix of $\frac{\partial L}{\partial h_n W}$, operator $*$ denote element-wise product.

In biological analogy, G is the feedback signal, the gradient for W is fine because the it do not involve signal from forward path. It only need the cell state h_n. However, the gradient for h_n is problematic in that a backward neural connection cannot know information from the forward path, say W. Asymmetric backpropagation now assume that $\frac{\partial L}{\partial h_n} = dot(G, B^T)$, which substitude W as something else.

2 Asymmetric Backpropagation With Only Dense Layers

To see whether this kind of things works, I tried MNIST for digit classification. But MNIST is so simple that everythings works.

Instead I trained some networks on CIFAR10 and then observe the difference.

2.1 Shallow Networks with Asymmetric Backpropatgation

The network here are merely dense layers, without convolutional layers. All nonlinearities is RELU, R denote W substituted by a random matrix, and optionally I can update them as W(marked as R*).

Also, hinge loss can be used instead of crossentropy. Adam is used for these training and results are collected after 100 epoch.

DS0:Input - dense512 - dense256 - dense10 - softmax - crossentropy.

DSR1:Input - dense512 - dense256 - dense10(R) - softmax - crossentropy.

DSR2:Input - dense512 - dense256(R) - dense10(R) - softmax -crossentropy.

DSR1-U:Input - dense512 - dense256 - dense10(R*) - softmax - crossentropy.

DSR2-U:Input - dense512 - dense256(R*) - dense10(R*) - softmax - crossentropy.

DS0-HL:Input - dense512 - dense256 - dense10 - softmax - hingeloss.

DSR1-HL:Input - dense512 - dense256 - dense10(R) - softmax - hingeloss.

DSR2-HL:Input - dense512 - dense256(R) - dense10(R) - softmax - hingeloss.

DSR1-U-HL:Input - dense512 - dense256 - dense10(R*) - softmax - hingeloss.

DSR2-U-HL:Input - dense512 - dense256(R*) - dense10(R*) - softmax - hingeloss.

The result are:

Table 1: Correctness on test set for dense layers, after 100 epoch

DS0	50.06%	DS0-HL	51.54%
DSR1	50.08%	DSR1-HL	48.83%
DSR2	44.19%	DSR2-HL	38.66%
DSR1-U	49.37%	DSR1-U-HL	48.82%
DSR2-U	50.49%	DSR2-U-HL	48.93%

It can be seen from the results that asymmetric backpropagation can achieve reasonable performance in shallow dense layer neural network. Stacking asymmetric layers make things worse.

2.2 Deeper Networks with Asymmetric Backpropatgation

Here more dense layers are used and there is some difference in the result.Again (R) marks the asymmetric random feedback in that layer, and (R*) marks the updating random feedback. As the network become deeper, the random feedback start to have severe impact on traininig process. Typically 100 epoch is not enough, so I also collect results after 200 epoch and put them in the brackect.

DS0:Input - dense512 - dense512 - dense256 - dense256 - dense10 - softmax - crossentropy.

DDR1:Input - dense512 - dense512 - dense256 - dense256 - dense10(R) - softmax - crossentropy.

DDR2:Input - dense512 - dense512 - dense256 - dense256(R) - dense10(R) - softmax -crossentropy.

DDR4:Input - dense512 - dense512(R) - dense256(R) - dense256(R) - dense10(R) - softmax -crossentropy.

DDR1-U:Input - dense512 - dense512 - dense256 - dense256 - dense10(R*) - softmax - crossentropy.

DDR2-U:Input - dense512 - dense512 - dense256 - dense256(R*) - dense10(R*) - softmax - crossentropy.

DDR4-U:Input - dense512 - dense512(R*) - dense256(R*) - dense256(R*) - dense10(R) - softmax - crossentropy.

DD0-HL:Input - dense512 - dense512 - dense256 - dense256 - dense10 - hingeloss.

DDR1-HL:Input - dense512 - dense512 - dense256 - dense256 - dense10(R) - hingeloss.

DDR2-HL:Input - dense512 - dense512 - dense256 - dense256(R) - dense10(R) - hingeloss.

DDR4-HL:Input - dense512 - dense512(R) - dense256(R) - dense256(R) - dense10(R) - hingeloss.

DDR1-U-HL:Input - dense512 - dense512 - dense256 - dense256 - dense10(R*) - hingeloss.

DDR2-U-HL:Input - dense512 - dense512 - dense256 - dense256(R*) - dense10(R*) - hingeloss.

DDR4-U-HL:Input - dense512 - dense512(R*) - dense256(R*) - dense256(R*) - dense10(R*) - hingeloss.

Table 2: Correctness on test set for dense layers, 200 epoch result in the bracket

DD0	49.06%	DD0-HL	51.67%
DDR1	38.82 (41.11)%	DDR1-HL	30.42 (39.34)%
DDR2	26.94 (31.60)%	DDR2-HL	29.27 (29.63)%
DDR4	27.7 (35.08)%	DDR4-HL	28.86 (26.36)%
DDR1-U	40.27 (45.70)%	DDR1-U-HL	40.46 (45.86)%
DDR2-U	35.35 (44.80)%	DDR2-U-HL	35.83 (45.43)%
DDR4-U	45.27 (42.18)%	DDR4-U-HL	39.65 (43.93)%

3 Poggio's Net with Asymmetric Backpropagation

In this section I used poggio's net, which is a shallow convolutional neural network with maxpooling and average pooling(Q Liao, JZ Leibo, T Poggio, AAAI 2016). The random feedback occurs in the dense layers. In the table below, correctness is showed together with result at 200/300 epoch in the bracket.

PG0: Input - Conv5*5*32/1 - Maxpool3*3/2 - Conv5*5*64/1 - Avgpool3*3/2 - Conv5*5*64/1 - Avgpool3*3/2 - dense128 - dense10 - softmax - crossentropy.

PGR1 : random feedback in last dense layer.

PGR2 : random feedback in last 2 dense layer.

PGR1-U: random updating feedback in last dense layer.

PGR2-U:random updating feedback in last 2 dense layer.

PGR1-HL : random feedback in last dense layer, hinge loss.

PGR2-HL : random feedback in last 2 dense layer, hinge loss.

PGR1-U-HL: random updating feedback in last dense layer, hinge loss.

PGR2-U-HL:random updating feedback in last 2 dense layer, hinge loss.

Table 3: Correctness on test set for Poggio's net, result after 200/300 epoch in the bracket

PG0	75.87%	PG0-HL	75.12%
PGR1	29.27 (40.65, 54.13)%	PGR1-HL	28.85 (40.38, 54.10)%
PGR2	33.84 (43.79, 43.87)%	PGR2-HL	30.93 (39.02, 40.55)%
PGR1-U	28.32 (53.77, 67.48)%	PGR1-U-HL	28.27 (50.89, 62.07)%
PGR2-U	27.44 (45.61, 57.95)%	PGR2-U-HL	31.52 (34.49, 43.31)%

From the result we can see that with random feedback in the dense layer the training is very hard. Typically more than 200 epoch is needed with Adam to achieve best result, though there is still large fluctuations even after 200 epoch. It's evident that stacking more random feedback layers prevent good result. Updating the random feedback can help, especially when we just have random feedback in the last layer. There is no significant difference between the hinge loss and crossentropy, most of time the crossentropy is better.

Using SGD works very well (68%) with (non updating) random feedback in the last layer, but given more random feedback dense layers, it failed(huge training/validation loss). Nestrov Momentum fails all the time with random feedback. Changing average pooling to max pooling in poggio's net can resolve this problem(with 2 random feedback layer), however the average pooling is very useful in improving the training result in poggio's net. Updating the random feedback in the same way as the true weight can also help avoid this problem.

4 Experiments on Untied Convolution

Here untied convolution means that for each receptive square in a single filter, the weight is independent. The weight sharing rule efficiently reduced the number of parameters, but this time we give up this advantage because we think this is biologically impossible. In the original convolution, the receptive squares share the same weight, which is believed to be not biological plausible. Cells in one position on the retina will by no means share weights with cells in other locations. The detailed implementation is to flatten the data and do convolution by an extremely large matrix dot operation. The matrix is deliberately designed that most entries is 0; It has been tested that using the same weight(weight should be flipped due to the fouier transformation), the result of my matrix dot and convolution is the same (there is some numerical issue so the error is 1e-7)

4.1 Untied convolution with random feedback only in the dense layer

Now testing the untied convolution on CIFAR-10, with Adam for 200 epoch. Adam is essential because the feedback signal for a single element of the weight is extremely small. In the original convolution the weight is shared for each location, which means that the feedback is the sum of a lot of locations. In the untied convolution we have to augment the signal or the training can hardly move during training. Adam is great because the signal magnitude do not matter, so we avoid this problem implicitly. The performance for the untied convolutional network, trained from scratch with possible random feedback, is shown below.

PG-Untie: Poggio's net with fully untied convolution layer.

PG-Untie-R1: with fully untied convolution layer, last dense layer has random feedback fixed.

PG-Untie-R2: with fully untied convolution layer, last 2 dense layer has random feedback fixed.

PG-Untie-R1-U: with fully untied convolution layer, last dense layer has random feedback updated.

PG-Untie-R2-U: with fully untied convolution layer, last 2 dense layer has random feedback and updated.

Table 4: Correctness on test set for Poggio's net

PG-Untie	64.90%
PG-Untie-R1	50.53%
PG-Untie-R2	38.70%
PG-Untie-R1-U	60.87%
PG-Untie-R2-U	34.61%

The untied convolution has a huge amout number of parameters that overfitting is very severe. It can be easily seen from the discrepancy between training loss and test loss(not shown in this summary); Updating the random feedback helps in some situation.

4.2 Untied convolution with random feedback in both dense layer and conv layer

Based on experiments of asymmetric untied convolution, it seems that training asymmetric untied layers from scratch is not practical. The loss function become intractible during training.

5 Gradient Clipping

When using asymmetric backpropagation, I tried to clip the gradient that involves the random feedback, ie $dot(h_n^T, G)$ is clipped.

I tried to clip it at [-1, 1], [-0.1, 0.1], but all these modifications ruin the training.