Weight Normalization in ConvPool-CNN-C Architecture

Xuanyi Liao xl2875, Linwei Gong lg3085
Columbia University

1. Abstract

As one of the reasons that help promote the advancement and progress for deep learning in recent years, batch normalization performs as an effective tool but it is not a perfect method without limitations, among which the key constraint is that its dependency on the mini-batch. The prime drawbacks are: it puts a lower limit on the batch size and makes batch normalization difficult to apply to recurrent connections in recurrent neural networks. Therefore, a new method, Weight Normalization[2] addressed by Tim Salimans et al. separates the norm of weight vector from its direction.

We present the algorithm implementation and experimental result analysis according to the paper *Weight Normalization: A Simple Reparameterization to Accelerate Training of Deep Neural Networks* and trained the ConvPool-CNN-C architecture based on the methodology addressed by the paper. We choose the CIFAR-10 dataset[1] which contains 60,000 RGB images to finish the model training part. After tuning the hyperparameters and finishing the training process, our model's error rate using weight normalization(17.88%) and weight normalization + mean-only batch normalization(18.33%) is proved to perform better than using batch normalization.

2. Introduction

The original paper, Weight Normalization: A Simple Reparameterization to Accelerate Training of Deep Neural Networks introduces a new method, weight normalization, instead of normalizing the mini-batch, to improve the training process and comprehensive performance in deep neural networks. For a standard ANN which is commonly trained by stochastic gradient descent for optimizing the weight vector and scalar bias of each neuron, different from earlier weight normalization approaches which only applies normalization after each step of stochastic gradient descent, the original paper proposes to reparameterize each weight vector by its fixing the Euclidean norm and perform stochastic gradient descent with respect to the related parameters. Besides the methodology and related algorithms, they carried out the experiments to validate and evaluate the performance of their method using different models in various application environments such as image recognition, generative modeling, and deep reinforcement learning. According to the experimental results, their method shows advantages over the applications mentioned above. Portability of implementation, low computational overhead and the elimination of dependencies between examples in mini-batch are the key properties of their work.

The goal of our project is to comprehensively learn the weight normalization algorithm addressed in the original paper, do the result replication part and finish the analysis of the experiments and results. Literature review, dataset introduction, algorithm implementation are also included in the following parts.

3. Related work

Mainly due to the pathological curvature problem, it can be a difficult problem to train deep neural network via first-order stochastic gradient descent. To conquer the problem, J.Martens et al.[6] addressed second-order optimization approach on the basis of Hessian-free method. R.B.Grosse et al.[7, 8] pre-multiply the cost gradient by using an inverse which aims to get an approximate natural gradient. The approximation inverse can be obtained by applying Cholesky factorization[8] or Kronecker-factored approximation[7]. However, these methods usually introduce much overhead. In addition, their optimization procedures are usually coupled with the preconditioning, and can not be easily implemented. S.Wiesler et al.[9] addressed the benefits of centered activations. They show that these transformations make the Fisher information matrix approximate block diagonal, and help the optimization perform better. Batch normalization [10] further standardizes the activation with centering and scaling based on mini-batch and includes normalization as a part of the model architecture. Ba et al. [11] calculated the layer normalization statics over all the hidden units within the same layers, so that the size of mini-batch can be limited. These approaches focus on normalizing the activation of the neurons explicitly.

There exist studies constructing orthogonal matrix in recurrent neural networks (RNN) [12, 13] by using reparameterization to avoid the gradient vanish and explosion problem. However, these methods are limited for the hidden to hidden transformation in RNN, because they require the weight matrix to be a square matrix. Other alternative methods [14, 15] focus on reducing the storage and computation costs by re-parameterization.

5. Dataset and Features

In our project we choose to use the CIFAR-10 dataset **Error! Reference source not found.**, there are 60000 images with the size of $32 * 32 * 3$. There are 10 classes which are airplane, automobile, bird, cat, deer, dog, frog, horse, ship, and truck. For each class, there are 6000 images. The dataset is divided into the training set with 50000 images and the test set with 10000 images.

Images are stored in a four-dimensional numpy array with the datatype of uint8, the first dimension represents the id of the image, the second and third dimensions represent the height and width of the image, the last dimension represents the channel number. The label is a list of numbers from 0 to 9. The number at the i-th entry indicates the label for the i-th image in the four-dimensional image array. Some sample images are shown below.

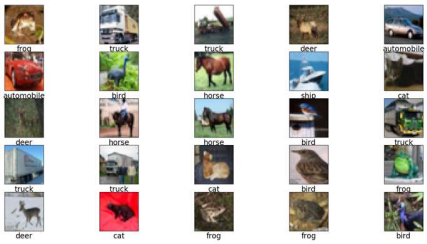

Figure 1

The image data is normalized before feeding into the neural network, every image is divided by 255.0 which is the maximum possible value of a pixel. After the division the pixel value becomes a double-precision floating-point number, we then convert it into a single-precision floating-point number in order to increase the training speed of our neural network.

6. Implementation

In our code, we first design the basic layers of our neural network, currently, the weight normalization is not implemented in Tensorflow, so we have to implement it with fundamental operations in Tensorflow.

A. Mean-only Batch Normalization

Mean-only Batch Normalization [2] is a special case of batch normalization, which can be combined with weight normalization. The idea is that the mean of mini-batch data is subtracted from the output of convolution or dense layers, which is the same as batch-normalization, but it is not divided by the standard deviation of the mini-batch. The only trainable parameter in the mean-only batch normalization is the offset term. We assume **w** is the weight matrix, **x** is the input of the current layer, then the activation t is:

$$t = \boldsymbol{w} \cdot \boldsymbol{x}$$
$$\tilde{t} = t - \mu[t] + b$$
$$y = \phi(\tilde{t})$$

Where $\mu[t]$ is the mean of minibatch of t. in the training process, $\mu[t]$ is replaced by the running average.

$$\mu_{new} = decay * \mu_{old} + (1 - decay) * \mu_{curr}$$

Where μ_{old} is the average of the previous time step, μ_{curr} is the average of the current mini-batch. By changing the value of decay, we can choose to weight more on current mean or accumulated mean. In our code, the decay is set to be 0.9, because the actual mean of the dataset is a fixed value, while the mean of mini-batches are unbiased estimate of actual mean, after many batches of training, the accumulated mean will be closer to the actual mean, so we want to weight more on accumulated mean, and perform little step update towards current mini-batch mean.

During the testing process, the running average of the mean will be used and its value will not change. The mean-only batch normalization has relatively less computation complexity than the batch normalization and causes less noise, due to the law of large numbers. They all lead to accuracy improvement.

B. Batch Normalization

It has been pointed out that part of the reason why training deeper neural network becomes more difficult is that the distribution of input for each layer changes little by little, and this phenomenon is called internal covariate shift [3].

Due to the existence of internal covariate shift, the deep neural network has to constantly adapt to non-stationary distributions which are harmful to training.

A common technique for data pre-processing is to normalize the data, which simply subtract the mean and divide the standard deviation. Normalization can help the neural network learn optimal parameters more quickly and avoid the explosion of loss.

A deep neural network can be viewed as a series of sub-network, the output of the previous layer is the input of the current sub-network, by performing normalization to every output of layers similar to data pre-processing, the training of sub-network would be faster, thus the whole network would benefit from it. The batch normalization transform algorithm is shown below.

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$
$$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$
$$\widehat{x_i} \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \qquad \text{// normalize}$$
$$y_i \leftarrow \gamma\widehat{x_i} + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Figure 2

The mini-batch mean is calculated by averaging all values in the mini-batch. The mini-batch variance is the average of the square of each value minus the mini-batch mean, which is a biased estimate of the true variance, we are not clear why the biased estimate is used here, maybe it is for stability consideration when the size of mini-batch is 1. The mini-batch is then normalized and scaled by γ and shifted by β, which are two trainable parameters.

In our code, the γ is initialized as 1 and β is initialized as 0, during the test phase, the batch normalization is implemented using running average of mean and variance, and they are fixed. During the training phase, the mean and variance of the current batch are extracted and they are used for calculating a running average of mean and variance.

C. 2-D Convolution

The convolution is an operation between kernel K and input I defined below:

$$s[i,j] = (I * K)[i,j] = \sum_m \sum_n I[m,n]K[i-m, j-n].$$

Figure 3

In two-dimensional situation, the kernel, usually a square matrix with odd size, is sliding across the whole image with constant stride, the result of convolution is the sum of pointwise multiplication if, without zero-padding, the size of the output will be smaller than the input, the output size of convolution is calculated using:

$$(W-F+2P)/S+1$$

where W is the input size, F is the kernel size, P is the padding, S is the stride.

In our code, the function of convolution accepts the input data, the number of kernels, the kernel size, padding mode, the stride, and the Boolean indicator of using mean-only batch normalization, batch normalization, and weight normalization.

According to the paper of weight normalization, the weight **w** can be parameterized as $g\frac{\boldsymbol{v}}{||v||}$ where g is a scaling parameter, **v** is the matrix that has the same size as **w**, and ||**v**|| is the Euclidean norm of **v**. it can be observed that the vector $\frac{\boldsymbol{v}}{||v||}$ is normalized and always has norm 1, so the Euclidean norm of **w** only depends on g, therefore it is called weight normalization. By decoupling the norm and direction of **w**, the stochastic gradient descent can converge faster [2]. It is also proposed to parameterize the scaling parameter g as e^s which theoretically more efficient, we did not implement this parameterization in our code.

The gradient of parameterized weight is shown below:

$$\nabla_g L = \frac{\nabla_{\mathbf{w}} L \cdot \mathbf{v}}{||\mathbf{v}||}, \qquad \nabla_{\mathbf{v}} L = \frac{g}{||\mathbf{v}||}\nabla_{\mathbf{w}} L - \frac{g\nabla_g L}{||\mathbf{v}||^2}\mathbf{v}$$

Figure 4

The computation is not related to the size of the mini-batch, so the overhead will be negligible.

The gradient of **v** can also be written in another format:

$$\nabla_{\mathbf{v}} L = \frac{g}{||\mathbf{v}||} M_{\mathbf{w}} \nabla_{\mathbf{w}} L, \quad \text{with} \quad \mathrm{M}_{\mathbf{w}} = \mathrm{I} - \frac{\mathbf{w}\mathbf{w}'}{||\mathbf{w}||^2}$$

Figure 5

M_w is a projection matrix that projects the gradient of v away from the gradient of weight, as the weight updates, the norm of **v** increases. When the gradient is noisy, **v** will increase quickly, then the scaling factor will decrease. If the norm of the gradient is small, then the norm of **v** stops increasing. Thus, this mechanism helps stabilize the norm and works well on a wide range of learning rate.

In our code, the norm of **v** is calculated by tf.nn.l2_normalize. The convolution layer with weight normalization will first be initialized before training, during the initialization, the mean and standard deviation of mini-batch are calculated and the bias is set to be -mean/std and g are set to be -1/std which makes the network start with each feature has zero mean and unit variance.

Different normalization techniques can be used in convolution layer, one thing need to mention is that batch normalization and weight normalization cannot be used at the same time. The output of the convolution is then fed into non-linear activation function which can increase the flexibility of the network.

D. Activation

Activation functions are non-linear and differentiable functions that usually follow the convolution layer and the dense layer. It can increase the non-linearity of the neural network so that the network can approximate complicated functions. In our code, we implemented three types of activation functions.

The first activation function is rectified linear unit ReLU function, and its form is very simple.

$$A(x) = max(0, x)$$

For the input x, if it is smaller than 0, then the activation will be 0, if it is greater than 0 then the activation is itself. The advantage of ReLU is that the gradient is easy to compute, 0 for negative input and 1 for positive input. Although the derivative at point 0 is discontinuous, we can pre-define it as 0. ReLU is commonly used in the convolutional neural network due to its simplicity, but there is one main problem called “dead neuron”, which means some neurons will never be activated if its output is always negative, then the neural network cannot fully utilize every neuron to learn the feature.

The second activation is Leaky ReLU. It is non-zero when the input is smaller than zero, the function is shown below:

$$A(x) = 1(x < 0)(ax) + 1(x >= 0)(x)$$

For the negative input, the slope is usually smaller than 1, usually 0.01. it increases the range of the ReLU activation function and solves the “dead neuron” problem. The curve of ReLU and Leaky ReLU is shown below.

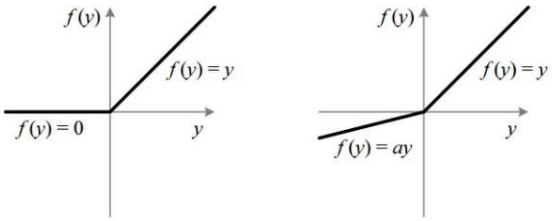

Figure 6

source: https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6

The last activation function is softmax, which is very common in classification. Softmax usually locates at the final dense layer, where the number of neurons is equal to the number of classes. Applying softmax to the dense layer can convert numbers into probabilities, although they are not the true probabilities. The function is shown below.

$$A(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}}$$

The x_i represents the i-th input, there are n inputs in total, and $A(x_i)$ is a number between 0 and 1, if we sum all $A(x_i)$s , then the result is equal to 1. We can view it as the probability of $P(input|class_i)$, there are 10 classes in our dataset, we simply pick the class which has the highest probability.

E. Adding noise

Adding noise to images is a kind of data augmentation technique, although it might be hard for a human to distinguish the image before and after adding noise, it is completely different for the computer. Since the computer can only see pixel values, any slight changes might affect the final output of the neural network. By adding noise, we can force the neural network to find the most important features and thus improve the robustness and reduce the overfitting

In our code, we implemented two types of noise, the first is the Gaussian noise, and the second is the uniform noise.

F. Dropout

An approximation method of the weighted sum network is the dropout. Dropout can reduce overfitting and provide a method of approximately combining exponentially many different neural network architectures efficiently[4].

The dropout process means to drop out neurons in a neural network, which temporarily disable or mask the presence of the neuron, the masked neuron will vanish from the network, its incoming and outgoing connections are blocked as shown below.

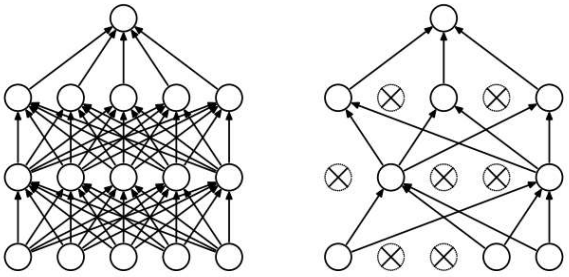

Figure 7

Each neuron in the neural network has fixed probability p independent from other neurons to be masked. According to the research, 0.5 is usually close to optimal for a variety of neural networks, and in our code, we choose 0.5, too.

G. Global Average Pooling

Global average pooling **Error! Reference source not found.** is proposed to minimize the total number of parameters in the neural network. Most of the parameters in a convolutional neural network belong to the dense layer, add global average pooling between the convolution layer and dens layer can significantly reduce parameters.

For a tensor with dimensions h*w*d, in our project is 32*32*3, the global average pooling calculates the average of pixels in h and w axis, in our project, the average of 1024 pixels, the output shape of the global average pooling is 1*1*d.

The global average pooling uses a single value to represent the whole image, which makes the convolution invariant to the region of interest, if the object in the training set is always on the left, and on the right in the test set, global average pooling can help eliminate the spatial variance and reduce overfitting.

In our code, the input is a 4-dimensional array with the shape [batch size, height, width, channels], we perform average on h and w axis and the result will become [batch size, 1, 1, channels].

H. Dense

In our code, the dense layer has a similar structure to the convolution layer, because both have trainable weights that weight normalization can be applied to. The dense layer accepts the input x, the number of neurons and activation function. It also accepts four Boolean indicators, for using weight normalization, using mean-only batch normalization, using batch normalization, testing or training.

If weight normalization is used, the norm of v will be calculated, during the initialization, b and g will be assigned to mean/std and -1/std

Finally, the result is fed into the activation function, if there is no available activation function, the result will be returned directly.

I. Full Convolution

It is actually a convolution operation with the kernel size 1*1. We simply reshape the input data and use the dense layer to calculate the result.

J. Structure

The network structure is shown below.

Raw input image 32*32*3
Normalization
Adding Gaussian noise $\sigma = 0.15, \mu = 0$
3*3*96 conv with leaky ReLU $\alpha = 0.1$
3*3*96 conv with leaky ReLU $\alpha = 0.1$
3*3*96 conv with leaky ReLU $\alpha = 0.1$
2*2 max pooling with stride 2
Dropout with p=0.5
3*3*192 conv with leaky ReLU $\alpha = 0.1$
3*3*192 conv with leaky ReLU $\alpha = 0.1$
3*3*192 conv with leaky ReLU $\alpha = 0.1$
2*2 max pooling with stride 2
Dropout with p=0.5
3*3*192 conv with leaky ReLU $\alpha = 0.1$
1*1*192 conv with leaky ReLU $\alpha = 0.1$
1*1*192 conv with leaky ReLU $\alpha = 0.1$
Global average pooling
Dense

Figure 8

7. Results

In our code, we use the Adam optimizer and the learning rate is 0.001, although in the paper, 0.003 is also tried, in our code, the training will not converge, so we choose 0.001.

The batch size for initialization is set to be 500 and the batch size for training is set to be 100. We choose a larger batch size for initialization because we want to reduce the variance of initial g and b in weight normalization. The number of epochs is 200, the same as the paper.

There are three hyperparameters that indicate whether or not to use weight normalization, batch normalization, and mean-only batch normalization, because weight normalization and batch normalization cannot be used simultaneously, mean-only batch normalization and batch normalization cannot be used simultaneously, there are five configurations for our convolutional neural network, which is shown below.

Normal
Weight normalization
Mean-only batch normalization
Batch normalization
Weight normalization+ Mean-only batch normalization

Figure 9

We first plot the training loss verses epochs, the figure is shown below.

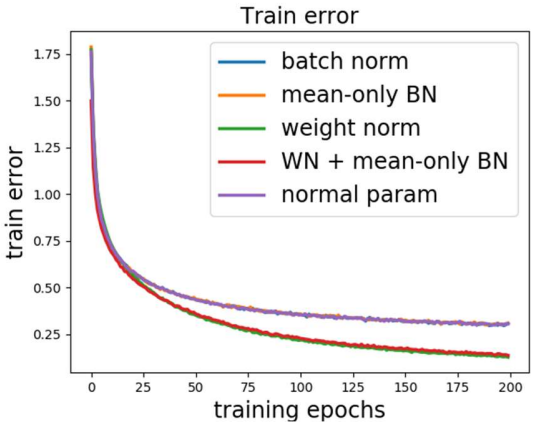

Figure 10

From the figure, we can see that the curve of weight normalization is on the bottom, and the curve of weight normalization + mean-only batch normalization is a little bit higher. The final training error of the red and green curve is below 0.25, while the other three are above 0.25.

The shape of our train error curve is different from the one in the paper which is shown below.

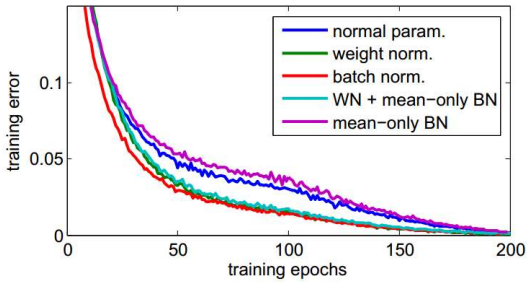

Figure 11

According to the paper, the optimizer is Adam with learning rate equal to 0.003 for the first 100 epochs, then for the remaining 100 epochs, the learning rate linearly decrease to zero.

We consider the reason why we produce a different result is that we did not use learning rate decay for the remaining 100 epochs, from the figure 11, the error keeps

going down after 100 epochs, but in figure 10, some stop learning or learn slower after 100 epochs.

The curve of the test error is shown below.

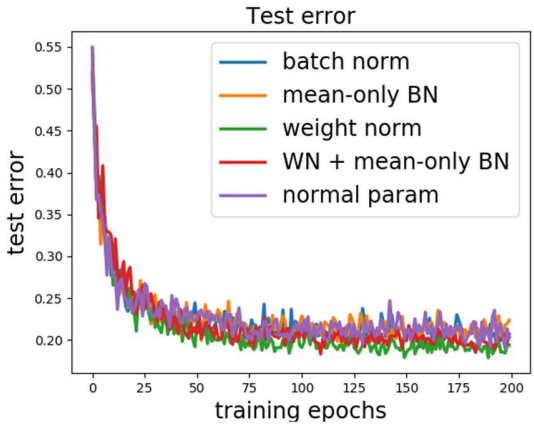

Figure 12

Although it is hard to distinguish all curves, it is still clear that the green line is on the bottom. We used the Savitsky-Golay filter in scipy to smooth the curve for better recognition, the window length is 51 and the polynomial degree is 10. The smoothed test error is shown below.

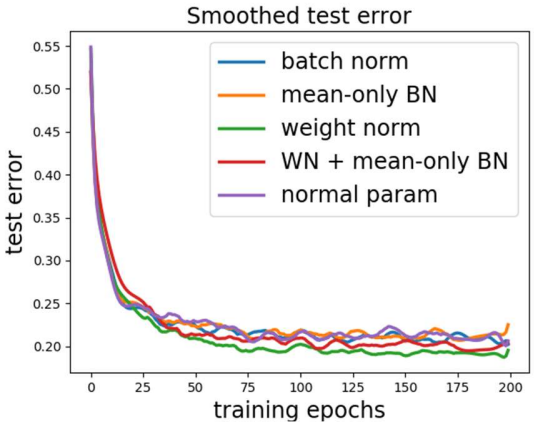

Figure 13

From the smoothed cure we can better distinguish each curve. The model with weight normalization achieves the lowest test error which is 17.88%, while according to the paper, the model with weight normalization achieves 8.46% test error rate, which is about 10% lower than ours.

	Our error	Paper's error
Batch norm	19.17%	8.05%
Mean-only BN	19.21%	8.52%
Weight norm	**17.88%**	8.46%
WN+MOBN	18.33%	**7.31%**
Normal	19.04%	8.43%

From the test error comparison, we can observe that both in our project and in the paper, the model with mean-only batch normalization has the highest test error, even higher than the model without any normalization. The model that uses weight normalization has a lower test error rate, which indicates that weight normalization can help improve the test accuracy and help model converge faster.

Finally, we compare the running time of each model, and the plot is shown below.

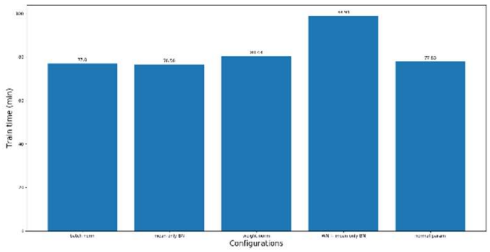

Figure 14

The model with weight normalization + mean-only batch normalization takes the longest time to train, about 100 minutes, the model with weight normalization takes 80 minutes to train and achieve the highest accuracy. The model with mean-only batch normalization takes the least time to train, 76.56 minutes.

The test err of weight normalization model is 1.29% lower than the batch normalization model. And takes 4.45% more time to train, which seems a fair trade.

One thing that needs to be noted here is that although the mean-only batch normalization model takes the least time to train and the weight normalization model takes a modest amount of time to train, the combination of mean-only batch normalization and weight normalization takes significantly longer time to train. We hypothesis that the combination of two increase the computation complexity by multiplication rather than linear addition

8. Conclusion and Future work

In this project, we implemented weight normalization for the convolution layer and dense layer in Tensorflow. Then we trained the ConvPool-CNN-C architecture on the CIFAR-10 dataset. We evaluate the performance of different normalization techniques and compare it with the result in paper [2] and proved the usefulness of weight normalization.

We realize the importance of learning rate decay, by comparing the train and test error in our project, we will implement it in the future.

There are three neural network models in the paper [2] that we do not implement in our project, they are VAE, DRAW, and DQN. In AlphaGo's algorithm, two neural networks are used to approximate the value and policy, what will happen if we apply weight normalization to this

network, will the performance be better or worse, we are eager to do an experiment on it in the future.

The code is in [16].

9. Contributions

Xuanyi Liao (50%)	Linwei Gong (50%)
1.Train the network 2. Write report 3.Performance analysis	1.Algorithm research and implementation 2.Write the report 3.Collect dataset

10. References

[1] Krizhevsky, Alex, and Geoffrey Hinton. *Learning multiple layers of features from tiny images*. Vol. 1. No. 4. Technical report, University of Toronto, 2009.

[2] Salimans, Tim, and Durk P. Kingma. "Weight normalization: A simple reparameterization to accelerate training of deep neural networks." *Advances in Neural Information Processing Systems*. 2016.

[3] S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *ICML*, 2015.

[4] Srivastava, N., G. Hinton, A. Krizhevsky, I. Sutskever, R. Salakhutdinov. "Dropout: A Simple Way to Prevent Neural Networks from Overfitting." Journal of Machine Learning Research. Vol. 15, pp. 1929-1958, 2014.

[5] Lin, Min, Qiang Chen, and Shuicheng Yan. "Network in network." *arXiv preprint arXiv:1312.4400* (2013).

[6] J. Martens and I. Sutskever. Training deep and recurrent networks with hessian-free optimization. In Neural Networks: Tricks of the Trade (2nd ed.). 2012.

[7] R. B. Grosse and J. Martens. A kronecker-factored approximate fisher matrix for convolution layers. In ICML, 2016. 1

[8] R. B. Grosse and R. Salakhutdinov. Scaling up natural gradient by sparsely factorizing the inverse fisher matrix. In ICML, 2015. 1

[9] S. Wiesler, A. Richard, R. Schluter, and H. Ney. Mean- ¨ normalized stochastic gradient for large-scale deep learning. In ICASSP, 2014.

[10] S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In ICML, 2015.

[11] L. J. Ba, R. Kiros, and G. E. Hinton. Layer normalization. CoRR, abs/1607.06450, 2016

[12] M. Arjovsky, A. Shah, and Y. Bengio. Unitary evolution recurrent neural networks. In ICML, 2016.

[13] S. Wisdom, T. Powers, J. Hershey, J. Le Roux, and L. Atlas. Full-capacity unitary recurrent neural networks. In NIPS, pages 4880–4888. 2016

[14] M. Denil, B. Shakibi, L. Dinh, M. Ranzato, and N. D. Freitas. Predicting parameters in deep learning. In Advances in Neural Information Processing Systems 26, pages 2148– 2156, 2013.

[15] Z. Yang, M. Moczulski, M. Denil, N. de Freitas, A. J. Smola, L. Song, and Z. Wang. Deep fried convnets. In ICCV, 2015.

[16] https://github.com/cu-zk-courses-org/e4040-2019fall-project-giao-lg3085-xl2875