Département Sciences du Numérique

Nuwa neural network framework

version 0.0.1

Zixing QIU

A new framework designed to provide a high-performance architecture for neural networks to solve partial differential equations.

Paris, 10 Dec 2021

Nuwa, a deep learning framework

Zixing QIU (INP-ENSEEIHT 3SN)

December 2021

Contents

Abstract

The aim of this project is to propose a new neural network framework to compute partial differential equations with higher performance. This project is still in early development stage, and the underlying development of fully connected neural network and a part of 2D CNN neural network has been completed. A dynamic back propagation graph is provided, it supports several optimizers, normalizations etc. The second phase will improve the project of the first phase to adapt it to the needs of computing partial differential equations. Finally, the Nuwa framework will be deployed as a C++ project and support CUDA and parallel computing. This article focuses on the structure and design philosophy of the first stage neural network framework, the computational principles and algorithm designs. The full code and updates are posted on GitHub

1 Introduction

The project is developed in Python (version 3) in both modular way and notebook type way in order to allow flexibility during the development and testing phases and also for the convenient of testing and understanding. The main files of the project are organized as follows:

1. **Nuwa v0.0.1.png:** Nuwa v0.0.1's architecture, Black bold unit represents it's a python class while green dashed line represents the relationship between these classes. For example, Dataset class parse the dataset to NN. Details are in the following section.
2. **Nuwa.ipynb:** A demo contains all the module of Nuwa framework, it contains 4 main parts of which the architecture will be represent later. It also provide a testing set, you could try your own dataset following the guide.
3. **README.md:** Project introduction and usage guide, contains the simple use of Nuwa framework and design concepts.
4. **dataloader.py:** Construction of *Dataset* class.
5. **demo.py:** Provide the same demo of two examples as Nuwa.ipynb.

6. **network.py:** Contains *NN* class for building neural network
7. **optimizer.py:** Construction of *Optimizer* class and related util
8. **requirement.txt:** The required dependencies of the project
9. **visualization.py:** *Visual* class is a class built for result plotting

For maximum showing the global sight of the project and its feasibility, **Nuwa.ipynb** provides two demos of Nuwa framework including four component of it and a testing set in order to test stability of the frame.

```
n = 300
x1 = np.random.uniform(0,1,n)
x2 = np.random.uniform(0,1,n)
const = np.ones(n)
eps = np.random.normal(0,.05,n)
b = 1.5
theta1 = 2
theta2 = 5
Theta = np.array([b, theta1, theta2])
y = np.array(b * const+ theta1 * x1 + theta2 * x2 + eps)
y = np.reshape(y,(-1,1))
X = np.array([const,x1,x2]).T
```

Let data set has 300 samples, assume that the model is linear and contains two parameters θ_1 and θ_2 and an intercept b. As result, feature data $x \in \mathbf{R}^3$ and feature matrix X has dimension $300 * 3$. We add a noisy $eps \sim \mathbf{N}(0, 0.05)$, the corresponding label $y = \theta^T X$

2 Nuwa architecture

Nuwa architecture has a total of 4 classes that deal with each take charge of one of the following 4 parts:

1. Import, pre-processing and distribution of data
2. Establishment of neural network, initialization and normalization of weights
3. Forward propagation and back propagation of the weights, and update the weights using optimization algorithms
4. Visualization of neural network output results and process data

This immediately gives us the easiest assignment to design a neural network framework. Nuwa's four main modules correspond exactly to these four requirements. Firstly the *Dataset* class is responsible for loading the data and assigning the training and test datasets. It is also responsible for allocation and shuffle in the mini batch type optimization algorithm. The *NN* class contains the elements needed to define each layer of the neural network, the activation function and batch normalization. When all the data and the neural network have been built, *Optimizer* will complete the whole training process. It contains forward propagation, dynamic graph backward propagation. During the training process it first initializes all the weights according to the constructed neural network, saves the calculation graph in the forward propagation, and finally updates the weights by the selected optimizer. It also provide loss functions to evaluate model. The results will be plotted by *Visual* with parameters in the training process.

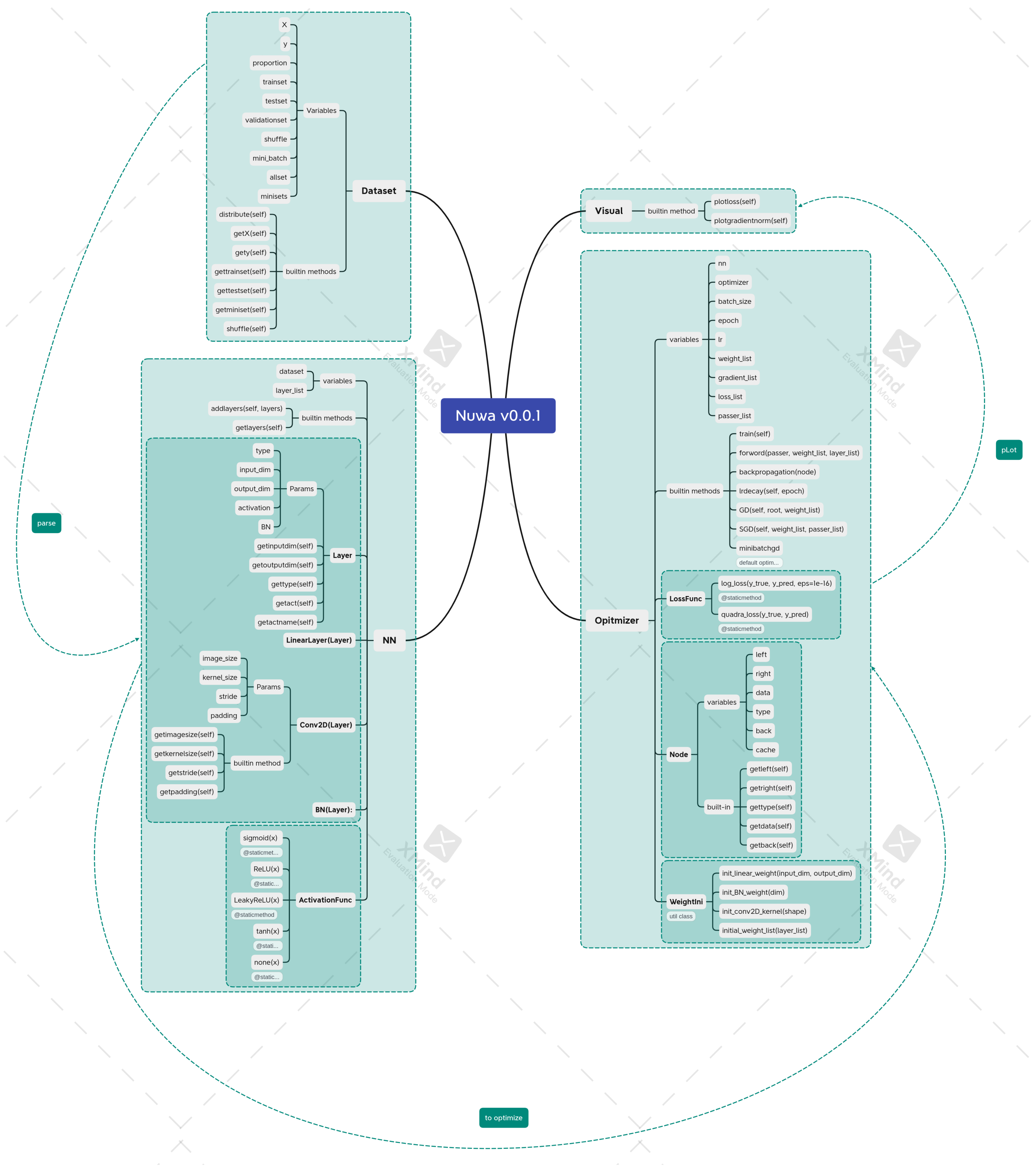

Figure 1: The basic framework of NUWA, *Dataset* loads the data, *NN* provides the neural network structure, *Optimizer* optimizes the training, and finally *Visual* visualizes the data.

2.1 Dataset

The constructor function of *Dataset* class is:

```
def __init__(self,X,y,proportion=0.8,shuffle=True, mini_batch=0):
```

X is feature data while *y* is the label corresponding to it. By changing the size of proportion, *Dataset* allocates the training set and test set sizes, and if shuffle is "True", then *Dataset* shuffles the order of the original dataset. Thus, during mini batch gradient descent, it can also dynamic change the order of data by using distribute function.

Dataset also proposes method for generating mini batch size dataset and returns a list of *Dataset* class, each also contains same mini batch.

2.2 NN

Inside the *NN* class, has the necessary components to form a neural network. It contains the *ActivationFunc* class, *Layer* class, where the *Layer* class is inherited by *LinearLayer*, *Conv2DLayer* and *BN*. There are several activation function can be chosen: "sigmoid", "ReLU", "LeakyReLU", "tanh" and "none" if no activation in the end.

The *Layer* constructor is given as:

```
    def __init__(self, type, input_dim, output_dim, activation, BN = False):
```

input_dim with output_dim, as their name shows, are the layer's input size and output size. they will be used when the weights are initialed. *NN* class can be constructed with an activation function. Also, the Activation function can be defined by *Layer* class. *NN* provides two ways to provide the activation function in order to be more flexible in changing the structure of the neural network. For example, it is still an open question to consider whether the BN should precede or follow the activation. Thus, in Nuwa, *Layer* class provides a much flexible method and options.

To make the API more concise, it is more convenient to use Layer's subclasses: *LinearLayer* and *Conv2DLayer*. Compare *Layer* and *Conv2DLayer*'s constructor:

```
class Conv2DLayer(Layer):
def __init__(self, input_size, kernel_size, stride, padding):
    self.type = "Conv2D"
    self.input_size = input_size
    self.kernel_size = kernel_size
    self.stride = stride
    self.padding = padding
    self.activation = "none"
```

Conv2DLayer defautly set self.type as "Conv2D" which avoid parse type to the layer. The same is true for *LinearLayer* where the subclasses of *Layer* only contain the properties of layer itself without any activation and normalization. The following provides a complete example of building a neural network. First *NN* receives the *Dataset* class entity object dataset for initialization and then passes in the list of Layers to build the network

```
    dataset = Dataset(X, y, mini_batch= 64)
    nn = NN(dataset)
    nn.addlayers(layer_list)
```

2.3 Optimizer

Optimizer is the most important class in the architecture of Nuwa, as the training process and testing process are provided by it:

1. lists: gradient list, weights list, loss list and passer list. All the list is accessible during model training
2. train(): including forword propagation and backpropagation
3. Node class allows every calculation in the backpropagation can be updated.
4. loss functions

Therefore, constructing an *Optimizer* instance should also include the above parts. It takes instance of *NN*, and should have an optimizer. In version 0.0.1 Nuwa provides "GD" for gradient descent, "SGD" for stochastic gradient descent and also "minibatchgd" for mini batch gradient descent. Note that if mini batch size equals 1, "minibatchgd" has the same result as thus in "SGD". Nuwa's default optimizer is "minibatchgd", so it has to set mini batch size when construct *Dataset* instance or set batch_size when construct optimizer instance. "epoch" refers to the number of times the optimizer needs to traverse the entire training set of samples. "decay_rate" is the decay rate when learning rate reduce with the times of epoch growth.

```
def __init__(self,nn,optimizer,batch_size=8,epoch=20000,lr=0.0001,decay_rate=0)
```

2.3.1 forword propagation

Before constructing how to forward propagation, it is necessary to first give the definition and role of Node, which is what makes Back propagation possible

```
def __init__(self, data: np.ndarray, type : str):
    self.left = None
    self.right = None
    self.data = data
    self.type = type
    self.back = None
    self.cache = None
```

Node class can be considered as node in binary tree, it has left child node and right child node who are *Node* class themselves. Each *Node* instance has data — their value during forward propagation while "type" is their corresponding type, it could be "weihgt", "data" (mini batch sample or whole sample X), or calculation like "+", "@" as inner product etc. "back" is the node's gradient updated during back propagation. Then finally, when it comes to complex gradient update rely on the middle value of the forward propagation, these kind of value can be stocked in the cache for later use.

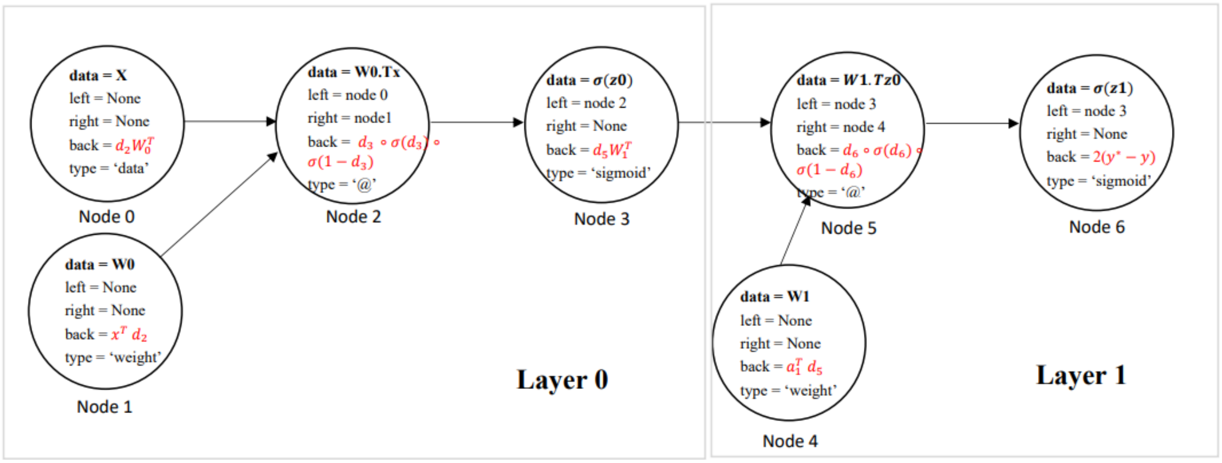

Figure 2: An example of how the node participate in the forward propagation, σ is sigmoid function, z_n represents inner product of layer n's weight with last layer's output before activation while a_n the output after activation.

Figure 2 is a vivid example of how node participates in the calculation of forward and back propagation. Assume neural network has 2 fully connected layers, the calculation can be noted as $a1 = \sigma(W_1^T \sigma(W_0^T X))$. W_0 and W_1 are the weights of each layer, σ is sigmoid activation function, note $z_0 = W_0^T X$, $a_0 = \sigma(z_0)$ etc. At the beginning, node 0's value is initialized as X, and Node 1 is a weight node whose value equals W_0, because layer 0 is a linear layer, Node 2 take the inner product of W_0 and X, set its left child node as node 0 and its right child node as node 1. The case of Layer 1 is exactly the same as Layer 0. This figure shows that forward propagation calculation graph is a binary tree with node 6 its root, every operations can be reduced to the union of two nodes or a single node, as the result, every node has at least one child node and have 2 child nodes maximum.

```
for i in range(layer_num):
    if layer_list[i].gettype() =='Linear':
        passer = passer@weight_list[weight_count].getdata()
        # append binary tree after inner product of weight and previous layer
        node = Optimizer.Node(passer,"@")
        node.left = passer_list[-1]
        node.right = weight_list[weight_count]
        passer_list.append(node)

        weight_count += 1

        if layer_list[i].getBN():
            node_cache = [passer, np.var(passer,axis = 0), np.mean(passer, axis=0 )]
            passer = (passer - np.mean(passer,axis=0))/np.sqrt(np.var(passer,axis=0))
            node = Optimizer.Node(passer,"normalization")
            node.cache = node_cache
            node.left = passer_list[-1]
            passer_list.append(node)

            node = Optimizer.Node(passer,"*scalar")
            node.left = passer_list[-1]
            node.right = weight_list[weight_count]
            passer_list.append(node)

            passer = passer + weight_list[weight_count+1].getdata()
            node = Optimizer.Node(passer,"+scalar")
            node.left = passer_list[-1]
            node.right = weight_list[weight_count+1]
            passer_list.append(node)
            weight_count += 2

        passer = layer_list[i].getact(passer)
        #append binary tree after activation function
        node = Optimizer.Node(passer,layer_list[i].getactname())
        node.left = passer_list[-1]
        passer_list.append(node)
```

The algorithm becomes apparently straightforward with the help of binary trees. In the training process, the weight list is first initialized, also including the gradient list and the passer list and the loss list. "passer" is the value flow along the forward propagation. For each layer in the neural network, the weights and the output of the previous layer are first calculated according to the corresponding operation, and the results are saved as node, and the left child node of this node is set as the output of the previous layer, and the right child node is set as the weight of the layer. If the layer has to be batch normalized, then first create a node for normalization, then calculate two weights γ and β, this simply add two passer node compare with no batch normalization. In the end, apply activation function on the previous' output, append the result into passer list as finish.

2.3.2 back propagation

During back propagation, note gradient of each node as d_n, for example, node 6's gradient is d_6. Let quadratic loss be our loss function, the "back" of node 6 is manually set as $2(y^* - y)$ where y^* is model predict label and y is sample label. Then back propagation can be expressed as a traversal of the binary tree over all nodes, that is, starting from root, all nodes update the gradient of their left and right child nodes with respect their values during froward and back propagation, another word, chain rule of gradient.

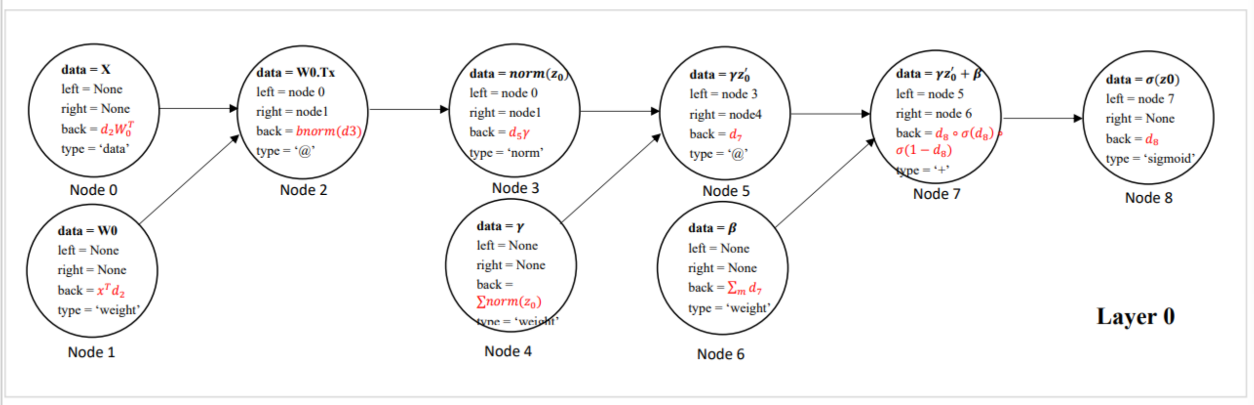

Figure 3: Another example of back propagation is batch normalization. One has to add two extra weights in the graph for update. Note the back propagation of *normalization* as *bnorm*() and z_0' as $norm(z_0)$

The most special of all operations is Batch normalization, because BN itself is not a layer but contains two weights, and the normalization process requires many basic operations. Figure 3 gives the calculation graph of BN, which we can still reduce to nodes on a binary tree. By adding γ and β to the weight list and updating them like any other weights, the gradient of normalization can be computed by one node.

```
def backpropagation(node):
    epsilon = 1e-8
    if node.getleft() is not None:
        if node.gettype() == "@":
            node.getleft().back = node.getback()@node.getright().getdata().T
            node.getright().back = node.getleft().getdata().T@node.getback()
        elif node.gettype() == "normalization":
            # cache = [x, sigma_beta^2, mu_beta]
            x = node.cache[0]
            sigma2 = node.cache[1]
            mu = node.cache[2]

            dl_dx_hat = node.getback()
            dl_dsigma2 = np.sum(dl_dx_hat,axis=0) * (x-mu) *
            -0.5*(sigma2+epsilon)**-3/2
            dl_dmu = np.sum(dl_dx_hat,axis=0) *
            -1/np.sqrt(sigma2+epsilon) + dl_dsigma2 *
            np.sum(-2*(x-mu),axis= 0)/x.shape[0]
            dl_dx = dl_dx_hat * 1/np.sqrt(sigma2+epsilon) +
            dl_dsigma2*2*(x-mu)/x.shape[0] + dl_dmu /x.shape[0]
            node.getleft().back = dl_dx
        elif ...

        Optimizer.backpropagation(node.getleft())
```

Based on the above binary tree algorithm, the backpropagation function is used as the basis for implementing the gradient update. *node* class is used as the input of the function, and the function provides the method for finding the gradient based on the type of the node, i.e., the operation. Take inner product and normalization as example, first, the gradient of child nodes are updated, after, the same process continues recursively for the left child node of the node. Note that there has no reason to recursively apply backpropagation function on the weight nodes , that's because, like the lower part of the above figures are weight nodes, they do not participate in operations other than their parent nodes, so they do not have child node either. When back propagation of BN, there are two ways to upgrade gradient of normalization: transform normalization into a series of arithmetic nodes that are recursively derived in backpropagation, while another method is used here, i.e., the intermediate process of normalization is stored in the cache variable of the node, so that all gradient updates can be done in a single node.

2.3.3 optimize algorithm

Nuwa provides three optimization algorithms: gradient descent, stochastic gradient descent and mini batch gradient descent. Since forward and back propagation have been implemented in the previous section, the optimization algorithm needs to address the way in which the gradient is updated. In this section we will only introduce mini batch normalization because the previous two can also be achieved by it.

```
# GD for every mini batch
minisets = self.nn.dataset.gettrainset().getminiset()
for j in range(len(minisets)):
    X_bar = minisets[j]
    self.passer_list[i].append(Optimizer.forword(X_bar.getX(),
    epoch_weight_list[j], layer_list))

    root = self.passer_list[i][j][-1]
    root.back = 2 * (-X_bar.gety() + root.getdata())

    weight, gradient = Optimizer.GD(self, root, epoch_weight_list[j])
    epoch_weight_list.append(weight)
```

As mentioned in section 2.1, *Dataset* class provides getminiset() method for auto generating mini batch size dataset. Therefore, this method is used in each epoch to shuffle the dataset and provide minisets.

The fourth and fifth line gets the data flow of the forward propagation, immediately afterwards, make the result of the passer list the root node, and do the back propagation. We use gradient descent on each weight of minisets.

In the mini batch gradient descent *Optimizer* modify learning rate by

```
    self.lr = 1 / (1 + self.decay_rate * iter) * self.lr
```

3 Result and Conclusion

The Nuwa framework so far provides operations on fully connected neural networks and partially convolutional neural networks, and a dynamic graph-based propagation algorithm is designed. The underlying code for the three optimization algorithms is also provided. The four classes correspond to the four parts of the neural network training. The figure shows the output of Nuwa in the test, the experiment provides a four layers network with each layer chooses sigmoid activation and also bach normalization, we construct this kind of network using following line:

```
layer_list = [NN.Layer('Linear',3,10,'sigmoid',BN=True),
NN.Layer('Linear',10,100,'sigmoid',BN=True),
NN.Layer('Linear',100,10,'sigmoid',BN=True),
NN.Layer('Linear',10,3,'none') ]
```

Use mini batch gradient descent as default optimizer, *Visual* shows that the whole training process converges quickly and the gradient does not show significant gradient vanish.

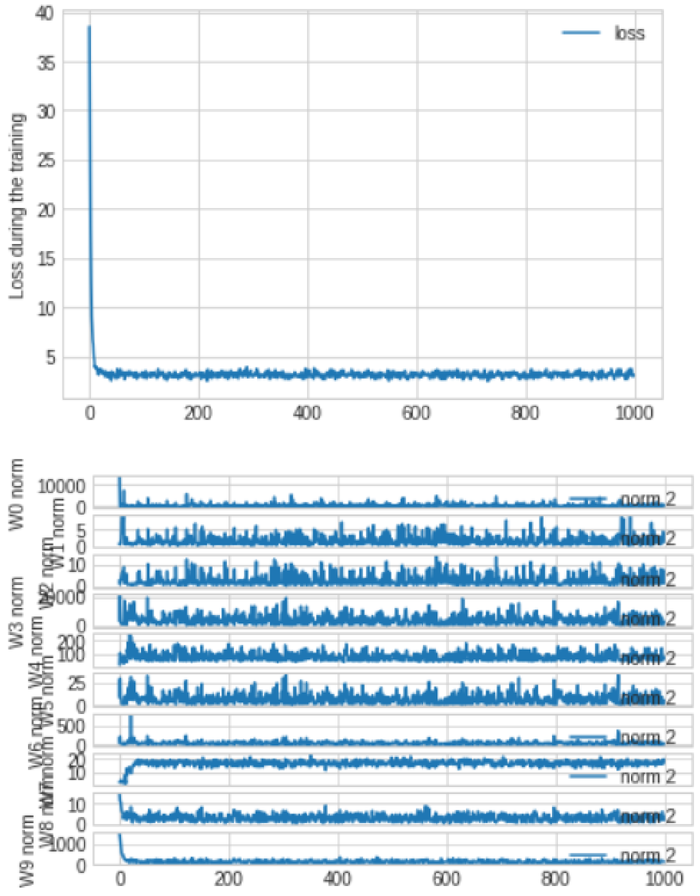

Figure 4: By using mini batch gradient descent and batch normalization, Nuwa gets a more stable fitting speed and converage rate.

However, due to the time constraints of the project, there were still many features in the first phase that needed to be further developed. For example, adding more optimizers, adding support for more loss functions, etc. The project is currently in the early stages of phase 2. The main purpose of the second phase is to continue to complete the Nuwa functionality while using its code to transform it into a framework similar to PINN and more suitable for solving partial differential equations, thus adding more physics constraints and APIs mainly from data-driven solution and data-driven discovery of partial differential equations domain. It should also make attention on whether the framework remains robust when it occurs the curse of dimensionality and focus on the numerical error when it comes to partial differential equation.

References

[1] Stuart J. Russell, Peter Norvig (2010) Artificial Intelligence: A Modern Approach, Third Edition, Prentice Hall ISBN 9780136042594.

[2] Hinton, Geoffrey; Sejnowski, Terrence (1999). Unsupervised Learning: Foundations of Neural Computation. MIT Press. ISBN 978-0262581684.

[3] G. Carleo, I. Cirac, K. Cranmer, L. Daudet, M. Schuld, N. Tishby, L. Vogt-Maranto, and L. Zdeborov ́a. Machine learning and the physical sciences. arXiv:1903.10563, (4):2773, Mar. 2019.

[4] Chu, I. Demir, K. Eichensehr, J. G. Foster, M. L. Green, K. Lerman, F. Menczer, C. O'Connor, E. Parson, and L. Ruthotto. White paper: Deep fakery—an action plan. Technical report, IPAM, 2020.

[5] Bengio, Y., Courville, A., Vincent, P. (2013). Representation learning: A review and new perspectives. IEEE transactions on pattern analysis and machine intelligence, 35(8), 1798-1828.

[6] Olga Russakovsky*, Jia Deng*, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg and Li Fei-Fei. (* = equal contribution) ImageNet Large Scale Visual Recognition Challenge. IJCV, 2015.

[7] M. Everingham, L. Van Gool, C. K. I. Williams, J. Winn, and A. Zisserman. The PASCAL Visual Object Classes (VOC) Challenge. IJCV, 2010.

[8] A. Krizhevsky, I. Sutskever, and G. Hinton. ImageNet classification with deep convolutional neural networks. In NIPS, 2012.

[9] URIA, Benigno, CÔTÉ, Marc-Alexandre, GREGOR, Karol, et al. Neural autoregressive distribution estimation. The Journal of Machine Learning Research, 2016, vol. 17, no 1, p. 7184-7220.

[10] GERMAIN, Mathieu, GREGOR, Karol, MURRAY, Iain, et al. Made: Masked autoencoder for distribution estimation. In : International Conference on Machine Learning. PMLR, 2015. p. 881-889.

[11] Goodfellow, I. J., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., ... & Bengio, Y. (2014). Generative adversarial networks. arXiv preprint arXiv:1406.2661.

[12] Zeiler, Matthew D and Fergus, Rob. Visualizing and understanding convolutional networks. In Computer Vision–ECCV 2014, pp. 818–833. Springer, 2014.

[13] Ioffe, Sergey and Szegedy, Christian. Batch normalization: Accelerating deep network training by reducing internal covariate shift. arXiv preprint arXiv:1502.03167, 2015.

[14] Nair, Vinod and Hinton, Geoffrey E. Rectified linear units improve restricted boltzmann machines. In Proceedings of the 27th International Conference on Machine Learning (ICML-10), pp. 807– 814, 2010.

[15] Xu, Bing, Wang, Naiyan, Chen, Tianqi, and Li, Mu. Empirical evaluation of rectified activations in convolutional network. arXiv preprint arXiv:1505.00853, 2015.

[16] Mordvintsev, Alexander, Olah, Christopher, and Tyka, Mike. Inceptionism : Going deeper into neural networks. http://googleresearch.blogspot.com/2015/06/ inceptionism-going-deeper-into-neural.html. Accessed: 2015-06-17.

[17] Kingma, Diederik P and Ba, Jimmy Lei. Adam: A method for stochastic optimization. arXiv preprint arXiv:1412.6980, 2014.

[18] Mikolov, Tomas, Sutskever, Ilya, Chen, Kai, Corrado, Greg S, and Dean, Jeff. Distributed representations of words and phrases and their compositionality. In Advances in neural information processing systems, pp. 3111–3119, 2013.

[19] Dumoulin, V. & Visin, F. A guide to convolution arithmetic for deep learning. https://arxiv.org/abs/1603.07285 (2016).

[20] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Going deeper with convolutions. CoRR, abs/1409.4842, 2014.

[21] M. Lin, Q. Chen, and S. Yan. Network in network. CoRR, abs/1312.4400, 2013.

[22] Trembacki, B. L., Mistry, A. N., Noble, D. R., Ferraro, M. E. & Mukherjee, P. P. et al. Editors' choice—Mesoscale analysis of conductive binder domain morphology in Lithium-ion battery electrodes. J. Electrochem. Soc. 165, E725–E736 (2018).

[23] Blair, S. C. Berge, P. A. & Berryman, J. G. Two-point correlation functions to characterize microgeometry and estimate permeabilities of synthetic and natural sandstones. Lawrence Livermore National Laboratory Report, 1–30 (National Lab., CA, USA, 1993).

[24] Mosser, L., Dubrule, O. & Blunt, M. J. Stochastic reconstruction of an oolitic limestone by generative adversarial networks. Transp. Porous Media 125, 81–103 (2018).

[25] Mao, X. Li, Q. Xie, H. Lau, R. Y. Wang, Z. & Smolley, S. P. "Least Squares Generative Adversarial Networks". 2017 IEEE International Conference on Computer Vision (ICCV), 2813–2821 (IEEE, Venice, 2017). https://doi.org/10.1109/ICCV.2017.304.

[26] Yoshida, Y. & Miyato, T. Spectral norm regularization for improving the generalizability of deep learning. https://arxiv.org/abs/1705.10941 (2017).

[27] Simonyan, K., & Zisserman, A. (2014). Very deep convolutional networks for large-scale image recognition. arXiv preprint arXiv:1409.1556.