Contents

1 Motivation

This library was initially designed to classify signals from electromagnetic Calorimeter detectors in PRad experiment at Jefferson Lab. Calorimeter detectors are widely used in modern particle and nuclear physics experiments, including both hadronic calorimeter (see [3]) and electromagnetic calorimeter (see [1]). Calorimeters are usually designed to completely stop particles that enter its body, inducing a full energy deposition of the incident particle. The deposited energy is converted into photons, these photons are then collected by Photo Multiplier Tubes (PMTs) attached on the calorimeter. Then the signals from these PMTs can be used to reconstruct the particle energy. By using proper design, calorimeters can also be used to reconstruct particle position information.

The particles that induced calorimeter signals in PRad experiment include photons, electrons and cosmic rays (muons). Photons and electrons are what PRad interests, cosmic rays are considered as a contamination. The typical signal patterns are shown in Fig. 1 and Fig. 2. The clear signal pattern difference makes convolutional neural network a very powerful tool to identify calorimeter signals. Using this software package, we reached an accuracy of more than 99%. This neural network package is also very suitable for general image process, for example the MNIST hand-written digits.

2 Tutorial

2.1 Overview

This software package needs C++11 standard, it was developed under CentOS 7.8 using GCC 4.8. It is purely written in C++, and is built from the ground. Users DO NOT need any additional external dependencies to run the program. Users can use this package to build convolutional neural network (CNN), multi-layer deep neural network (DNN) and Auto-encoder neural networks. Concrete examples for different types of neural networks

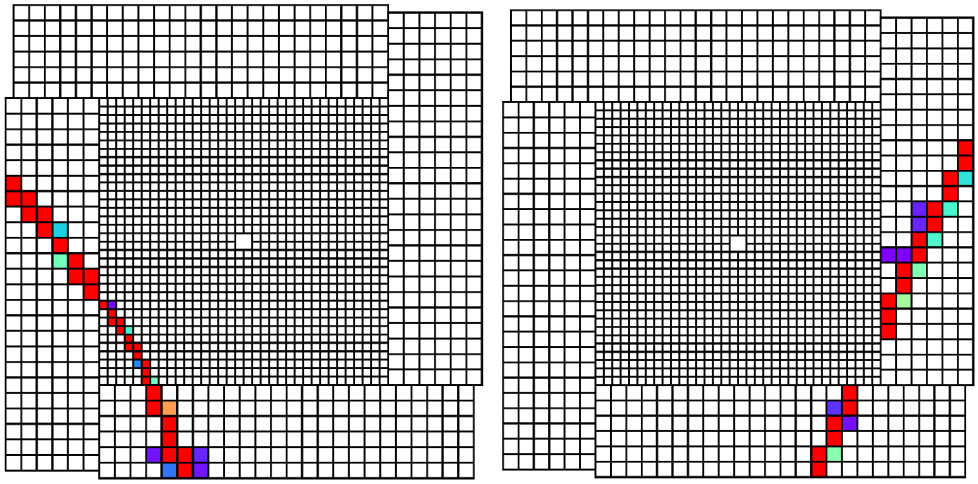

Figure 1: Cosmic signals in calorimeter. The square blocks are calorimeter modules. There are two types of modules: crystal module (inner region) and PbGlass module (outer region).

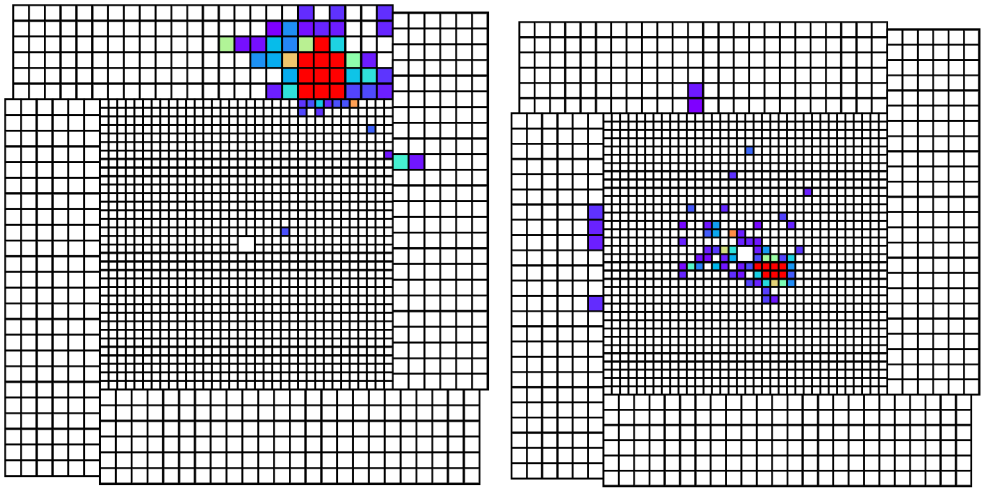

Figure 2: Electron and photon signals in calorimeter. Calorimeter cannot distinguish electrons from photons, the patterns from the two type of particles are similar.

have been given inside the package. A very minimum knowledge of neural networks is required for reading this tutorial.

Three basic neural network (NN) concepts were employed in this package, *Neuron*, *Layer* and *Network*. In a nutshell, *Layers* house *Neurons*, *Networks* are composed of *Layers*. The principal unit is *Layer*, which is responsible for taking the output of its neurons and form images, and then pass these images to its neighbouring layers (in both forward and backward directions). *Layers* are divided into two categories, 1 dimensional (1D) layer and 2 dimensional (2D) layer, based on the dimension of its output in each sample. For example, fully connected (FC) layers are 1D, since the neurons inside an FC layer are arranged in a column matrix format, and the output is in a vector format. CNN and Pooling layers are 2D, since the output of these layers are in a 2D matrix format.

A purely abstract base *Layer* class was designed to uniformly refer to any type of layers. This class consists of purely virtual functions and was designed to hold the pointers of any type of concrete layers. The Matrix operation has also been implemented in the package. Parallel computing was implemented using C++11 thread, one can turn on parallel computing for training, classifying in neural networks; one can also turn on the parallel computing for large Matrix operation. However, if one does not have enough computing cores, it is not recommended to turn on both of them, the overhead of thread allocation will slow down the program running speed. By default, NN parallel computing is ON, Matrix parallel computing is OFF. On Jefferson Lab computing farm, we used computing nodes with 128 logic cores each, and the program speed roughly scale with the number of computing cores in use.

2.2 Use of the Code

A parameter list struct was defined to manage parameters for layer construction:

```
1  struct LayerParameterList{
2      LayerType _gLayerType; // cnn, pooling, fc, input, output
3      LayerDimension _gLayerDimension; // 1D or 2D
4      DataInterface *_pDataInterface;  // feed data to network
5
6      size_t _nNeuronsFC;  // neuron number for FC, (CNN and Pooling ignores
       this one)
7      size_t _nKernels; // number of kernels for CNN and pooling
8      std::pair<size_t, size_t> _gDimKernel; // kernel Dimension
9
10     float _gLearningRate; // hyper parameter for learning rate
11
12     bool _gUseDropout;  // switch on/off drop out
13     int _gdropoutBranches; // max number of drop out branches
14     float _gDropoutFactor; // how many neurons to drop out
15
16     Regularizatoin _gRegularization; // switch on/off regularization
17     float _gRegularizationParameter; // hyper parameter for regularization
18
```

```
    ActuationFuncType _gActuationFuncType; // sigmoid, tanh, relu

    TrainingType _gTrainingType; // new training or resume training
};
```

The above struct may contain more parameters than an actual layer needs, for these un-needed parameters, the layer simply ignores them.

- **DNN Network Construction**

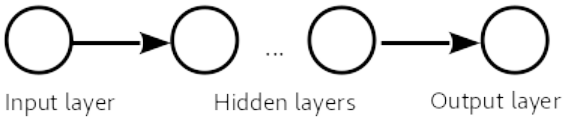

Figure 3: Conceptual structure of deep neural network.

The structure is shown in Fig. 3. It includes:

1) The interface for data process:

```
DataInterface *data_interface = new DataInterface(const char* path, ...,
    LayerDimension LD, std::pair<int, int> dim, size_t batch_size );
```

2) Input layer:

In this package, an input layer is simply for feeding the data to the network, it does not participate in training process. Only hidden layers and output layer participate in training, in other words the weights and bias in these layers need to be updated during training.

```
// setup parameters
LayerParameterList p_list0(LayerType::input, LayerDimension::_1D,
    data_interface, 0, 0, std::pair<size_t, size_t>(0, 0), 0, false, 0, 0,
    Regularization::Undefined, 0, ActuationFuncType::Undefined,
    TrainingType::NewTraining);
// construct layer
Layer* layer_input = new ConstructLayer(p_list0);
// initialize the layer
layer_input->Init();
```

3) Hidden layers

```
// setup layer parameter
LayerParameterList p_list3(LayerType::fullyConnected, LayerDimension::_1D,
     data_interface, 10, 0, std::pair<size_t, size_t>(0, 0), 0.06, true, 2,
     0.5, Regularization::L2, 0.01, ActuationFuncType::Relu, TrainingType::
    NewTraining);
// construct layer
Layer *l3 = new ConstructLayer(p_list3);
// setup its left side neighbour
l3->SetPrevLayer(layer_input);
// initialize
l3->Init();
```

For hidden and output layers, one needs to setup its preceding neighbour layer before initialization, because the initialization process rely on the information from its preceding neighbour to automatically setup weight Matrix dimensions.

One can construct as many hidden layers as needed, in the similar way like above. The number of hidden layers is unlimited.

4) Output layer

```
// setup parameters
LayerParameterList p_list_output(LayerType::output, LayerDimension::_1D,
    data_interface, 2, 0,  std::pair<size_t, size_t>(0, 0), 0.06, false, 0,
     0., Regularization::L2, 0.01, ActuationFuncType::SoftMax, TrainingType
    ::NewTraining);
// construct layer
Layer* layer_output = new ConstructLayer(p_list_output);
layer_output -> SetPrevLayer(l3);
// initialize
layer_output -> Init();
```

5) Connect layers

One need to connect the constructed layers together to make a network. The connection means setting up each layer's preceding and following layers. The preceding layer has already been setup during construction, so only the following connection is needed.

```
    //l1->SetNextLayer(l2);
    //l2->SetNextLayer(l3);
    l3->SetNextLayer(layer_output);
```

The above 5 steps conclude the construction of DNN networks.

- **CNN Network Construction**

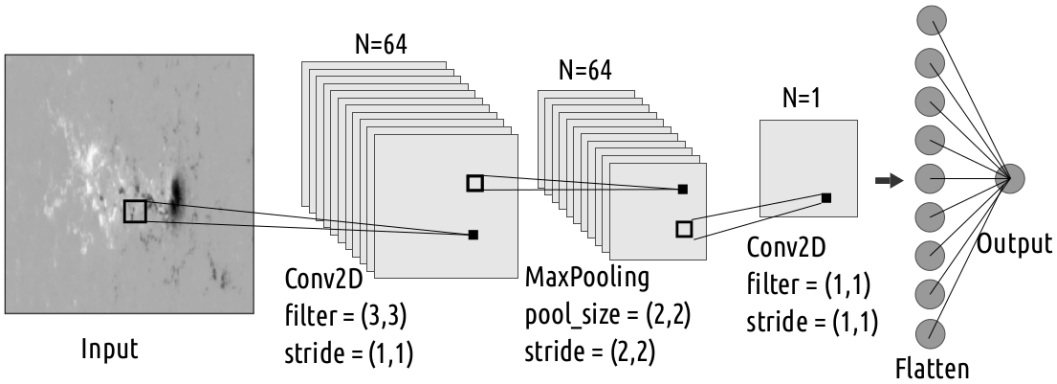

Figure 4: Simple structure of convolutional neural network. This plot is taken from [7].

The structure is shown in Fig. 4. For CNN, the input layer and output layer construction is the same with DNN. CNN layer and pooling layer are also constructed in a similar way:

1) CNN layer

```
// setup parameters
LayerParameterList p_list1(LayerType::cnn, LayerDimension::_2D,
    data_interface, 0, 3, std::pair<size_t, size_t>(9, 9), 0.06, false, 0,
```

```
    0.5, Regularization::L2, 0.01, ActuationFuncType::Relu,
    resume_or_new_training);
// construct layer
Layer *l1 = new ConstructLayer(p_list1);
l1->SetPrevLayer(layer_input);
// initialization
l1->Init();
```

2) Pooling layer

```
// setup parameters
LayerParameterList p_list2(LayerType::pooling, LayerDimension::_2D,
    data_interface, 0, 3, std::pair<size_t, size_t>(3, 3), 0.06, false, 0,
    0., Regularization::L2, 0.01, ActuationFuncType::Relu,
    resume_or_new_training);
// construct layer
Layer *l2 = new ConstructLayer(p_list2);
l2->SetPrevLayer(l1);
// initialize
l2->Init();
```

At present, Max pooling and Average pooling method are implemented. One need to make sure that the number of kernels used for pooling must equal to the number of kernels in its preceding CNN layer.

The connection of layers is also in the same way with DNN. However, there is one thing need to be pointed out: CNN and pooling are 2D layers, FC layers are 1D layers. When connecting layers with different dimension, the program will automatically perform Vectorization() and Tensorization(), basically transforming 1D matrix to 2D matrix or vice versa. At present, 1D layers cannot be followed by 2D layers, 2D layers can be followed by both 2D and 1D layers. For example, in the forward direction:

$$\text{CNN} \rightarrow \text{CNN} \rightarrow \text{Pooling} \rightarrow \text{CNN} \rightarrow \text{Pooling} \rightarrow \text{FC} \rightarrow \text{FC}$$

type of connection is allowed.

$$\text{FC} \rightarrow \text{CNN/Pooling}$$

type of connection is not implemented. Both types are allowed in the backward direction.

The CNN kernel stride is currently hard coded to be 1, with the assumption that any location of the input can have sensitive information. A configurable CNN stride is planned in future updates.

Zero padding is enabled by default in CNN and pooling layers.

• **Auto-encoder Network Construction**

Auto-encoder NN is constructed similarly like DNN. The difference is only in prediction: for DNN, one use the output layer for prediction; for Auto-encoder, one use the bottle neck layer in the middle for prediction.

Auto-encoder is un-supervised learning, one need to replace the label matrix in DataInterface class with data sample itself. This can be easily approached in the program.

• **Data Interface**

The data interface class currently organize data into Matrix format. **One need to read and understand the code to make sure the input and output data format are consistent with one's requirement. A more general code for data process is NOT planned, since it is so easy for users to customize the DataInterface class to load any format of data using C++ programming language.**

Users only need to construct layers to form networks, the rest part including training, prediction etc are already taken care of in the package. **However, users are encouraged to modify the code as much as possible.**

3 A Concrete Application Example

A simple, concrete example has been provided in the package, it could help users get familiar with the program quickly. The data is placed in *simulation_data* directory. It is a simulated data set based on PRad experiment. It is NOT the PRad data, since publicize the experiment data needs permission from the collaboration. And the format of the data has been made very different with experiment data. The pattern of the simulated data is shown in Fig. 5.

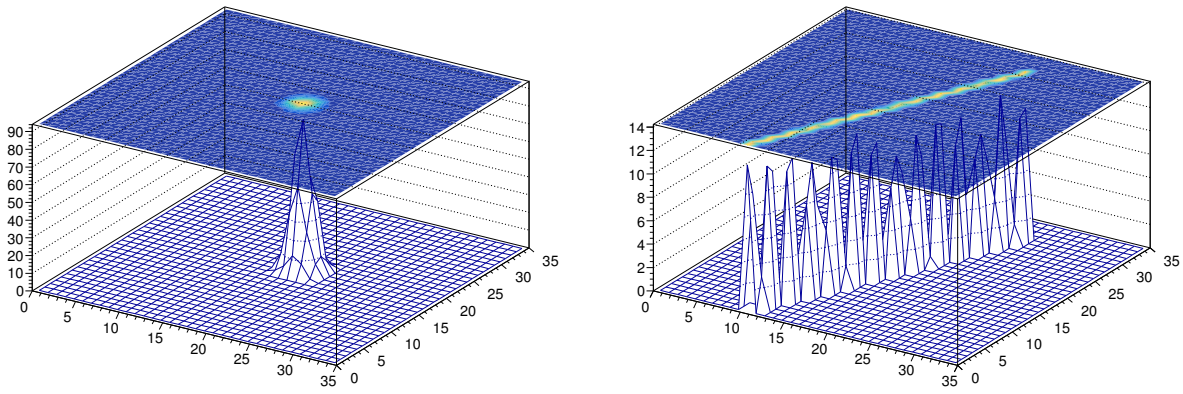

Figure 5: Zoomed in plots for simulated electron (left) and cosmic (right) signals.

Different types of neural networks and optimization algorithms were tested, the results are shown in the following figures respectively. All these tests were based on the data set shown in Fig. 5.

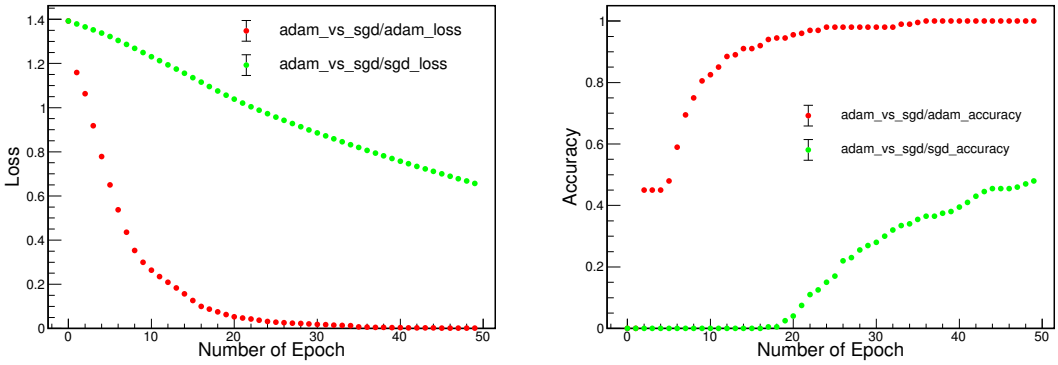

Figure 6: Loss and accuracy test results from a CNN → CNN → FC structure neural network model. The red color is from Adam optimizer [4], the green color is from SGD (Stochastic Gradient Descent) method. All other contributions are set exactly the same. The two tests have the same weight initialization, same learning step size, same regularization factor etc. The optimization method is the only difference. The left plot is loss comparison, the converge speed from Adam optimizer is indeed better than SGD; The right plot is accuracy during training, the first two epochs all have zero accuracy in both methods, the Adam optimizer reached a better result than SGD in later epochs.

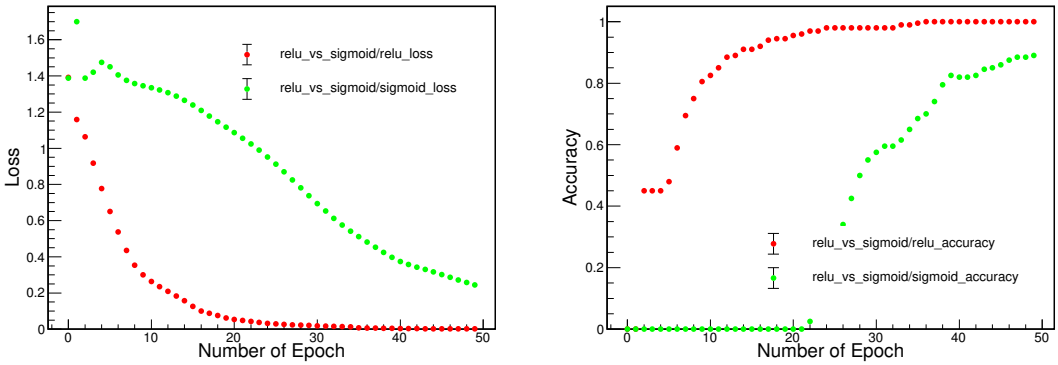

Figure 7: Loss and accuracy test results from a CNN → CNN → FC structure neural network model. The red color is RELU neurons, the green color is Sigmoid neurons. All other contributions are set exactly the same. RELU neurons outperformed Sigmoid neurons in this classification problem. In this test, the Adam optimizer was used.

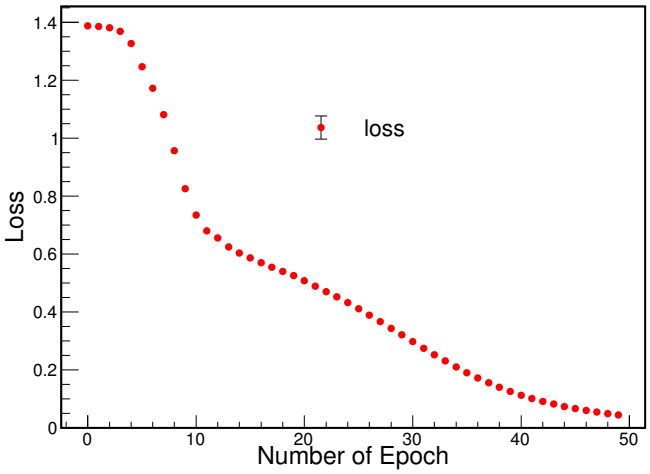

Figure 8: Loss from a redundant neural network: {Image → CNN → pooling → CNN → pooling → FC → FC}. For the classification problem shown in Fig. 5, the structure of the network in this test is unnecessarily complicated, the accuracy quickly reached 1 during the test, which indicated that over-fitting occurred in the training. The learning rate was set to 0.06, the regularization factor was set to 0.005. Under this setting, the SGD optimizer quickly encountered Gradient vanishing problem and stopped learning, the Adam optimizer managed to continue learning.

3.1 Preliminary Test on MNIST Hand Written Digits

A very preliminary test was performed on MNIST data for sanity check. This test used 30 computing threads, and it takes about 2 minutes to finish 60000 samples. A few samples of MNIST data was shown using ROOT in Fig. 9. A very rough test results was given in Fig. 10.

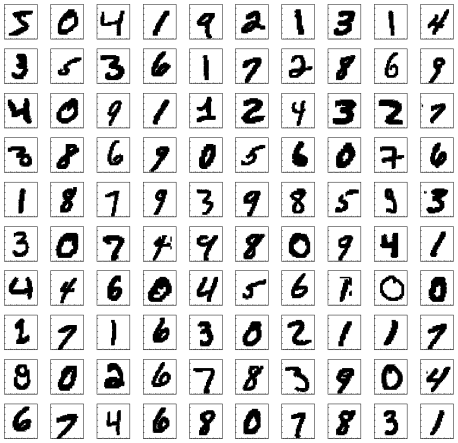

Figure 9: MNIST digits.

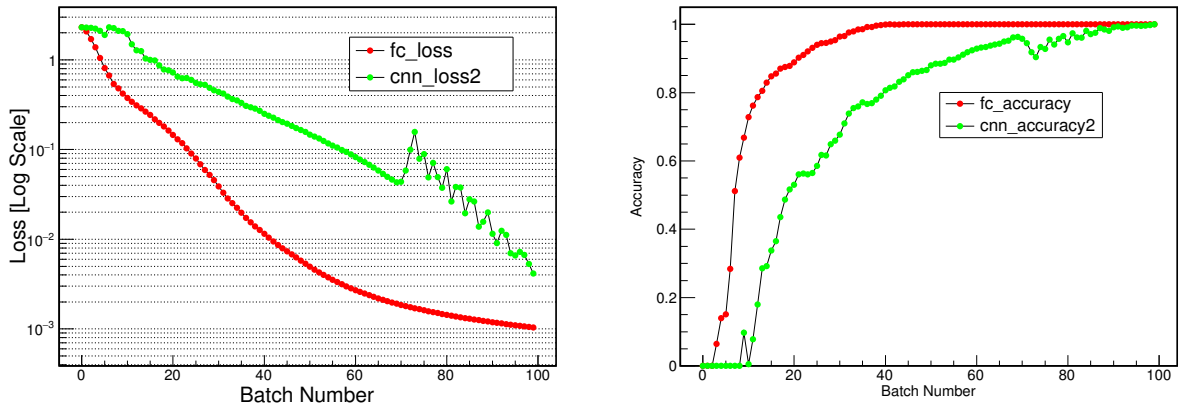

Figure 10: Loss and accuracy test results on a MNIST subset. The FC used one hidden layer and one output layer. The CNN used the same network structure as described in Fig. 8.

4 Future Improvements

1) Weight Initialization

The neural network performance (converge speed, etc) seems rely on weight initialization quite a bit, even though the **Xavier** initialization [2] was implemented for logistic and hyperbolic neurons, and the **Kaiming He** initialization [5] was implemented for RELU neurons.

This performance-versus-initialization correlation was very obvious when using Stochastic Gradient Descent (SGD) optimization method. Implementing Adam optimization method greatly reduced the correlation. But still, even with Adam optimization, occasionally one still get slow training speed, see Fig. 11.

The slow converging speed issue can be easily noticed in the early training stage. Thus, if one encounter slow converge speed, one may want to kill the program and re-run it, which can re-initialize all the weight matrices.

This correlation needs to be carefully studied in the future, for example implement the Batch Normalization method reported in [6].

2) Different Types of Neurons

Other types of neurons need to be implemented, like leakage-RELU, softsign etc...

3) More Optimization Methods

Other optimization methods need to be implemented, like AdaGrad, RMSProp, implicit SGD, Newton Method (second order derivative) etc...

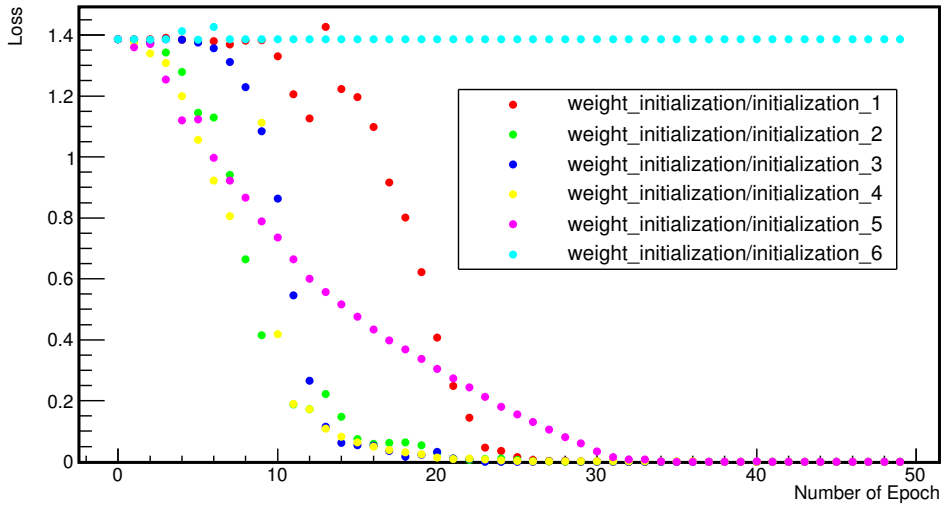

Figure 11: The loss during training of the redundant Neural Network (model described in Fig. 8) using different weight initializations (other contributions such as learning step size, regularization factor etc are all kept the same). In this test, the Adam optimizer was used. Weights were initialized randomly using the method presented by Kaiming He et al (see [4]). Occasionally, we encounter weight initializations with very slow converging speed, like the one shown in Cyan color (initialization_6). The cyan color is still learning, only at a very slow speed. One can use a simpler model to mitigate this issue.

References

[1] G. Mavromanolakis. *Quartz fiber calorimetry and calorimeters*. 2004. arXiv: `physics/0412123 [physics.ins-det]`.

[2] Xavier Glorot and Yoshua Bengio. *Understanding the difficulty of training deep feedforward neural networks*. `{http://proceedings.mlr.press/v9/glorot10a/glorot10a.pdf}`. 2010.

[3] Jose Repond. *Development of Particle Flow Calorimetry*. 2011. arXiv: `1110.2121 [physics.ins-det]`.

[4] Diederik P. Kingma and Jimmy Ba. *Adam: A Method for Stochastic Optimization*. 2014. arXiv: `1412.6980 [cs.LG]`.

[5] Kaiming He et al. *Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification*. 2015. arXiv: `1502.01852 [cs.CV]`.

[6] Sergey Ioffe and Christian Szegedy. *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift*. 2015. arXiv: `1502.03167 [cs.LG]`.

[7] Shamik Bhattacharjee et al. *Supervised convolutional neural networks for classification of flaring and nonflaring active regions using line-of-sight magnetograms*. 2020. arXiv: `2005.13333 [astro-ph.SR]`.