Review of Three Neural Network Journals & Weight Reparameterization Implementation of CNN's

Anonymous Authors[1]

Abstract

This document provides a a critical review of three (3) published papers, two of which is with regards to the structure and application of Graph Neural Networks (GNN's). The first paper is a review of using a deep convoluted GNN to model COVID-19 transmission rates across country borders. COVID-19, also known as SARS-CoV-2 went from a local outbreak to a large-scale international pandemic in 2020. In a continually impacting pandemic, GNN's can be used to predict transmission rates in certain areas around the globe. The third research paper touches upon analyzing the performance of graphical neural networks and determine the underlying conditions that make GNN's practical and applicable over non-geometric neural networks. The second research paper describes a very practical approach to reparameterizing deep neural networks for improved optimization, that can also be used in graphical applications as well. The following section has the respective implementation of that reparameterization with the corresponding analysis, discussion, evaluation, and next steps to be taken.

1. Transfer Graph Neural Networks for Pandemic Forecasting [1]

Transfer Graph Neural Networks for Pandemic Forecasting starts with a brief overview of the pandemic that has ravaged almost every country around the globe. There is a main difference that is covered early that indicate the authors use of natural human tendencies to move around a certain area. The authors indicate that, while proxy movement within a certain region does contribute greatly to the transmission of COVID-19, the focus of the application of this GNN is to use the human mobility from one localized region to another as natural graph edges. This gives rise to the recent interest of relation learning techniques such as graph structures.

First, let us start by defining the structure of GNN's: $G = (E, V)$ where G is an graph with $E =$ directed edge weights (can contain self-loops and therefore cycles) and V = vertex. For each period in time (unit = t = day) there is a corresponding graph G^t such that an array exists $G^{(t)} = G^{(1)}, G^{(2)}, ..., G^{(N)}$. The main benefit of utilizing graph structures is the fact that each vertex in the graph represents a region with a case infection rate c_u^t where $u =$ example region node label @ time t. It is important to note that when conducting the prediction of infection rate, the paper takes the past d days relative to t to account for the irregularity in reporting infection cases. Moreover, the graphical edges, as priorly defined, will be defined by the transfer of people from one region to another.

Subsequently, this structure naturally leads to a linear combination:

$$\boldsymbol{x}_u^{(t)} = (c_t^{(t-d)}, ..., c_u)^T \in \mathbb{R}^d$$

where T is the linear transpose.
This leads us to the linear combination $\boldsymbol{z}_u = (A_{j,u}\boldsymbol{x}_j)$ and

$$\boldsymbol{A}^{(t)}\boldsymbol{X}^{(t)} = \begin{bmatrix} \boldsymbol{A}^{(t)}_{(1,1)} & \boldsymbol{A}^{(t)}_{(2,1)} & \cdots & \boldsymbol{A}^{(t)}_{(n,1)} \\ \boldsymbol{A}^{(t)}_{(1,2)} & \boldsymbol{A}^{(t)}_{(2,2)} & \cdots & \boldsymbol{A}^{(t)}_{(n,2)} \\ \vdots & \vdots & \vdots & \vdots \\ \boldsymbol{A}^{(t)}_{(1,n)} & \boldsymbol{A}^{(t)}_{(2,n)} & \cdots & \boldsymbol{A}^{(t)}_{(n,n)} \end{bmatrix} \begin{bmatrix} \boldsymbol{x}_1^{(t)} \\ \boldsymbol{x}_2^{(t)} \\ \vdots \\ \boldsymbol{x}_n^{(t)} \end{bmatrix}$$

Using the two equations provided, $\boldsymbol{z}$ can be seen as the average estimate in new cases in a region u. Moreover, the AX^T can represent the new cases within region u.

A specific type of neural network Message Passing Neural Network (MPNN) is used to represent topological maps of a country - the network consists of regional aggregate layers where each layer has the aforementioned graph structure. Each graph feature from the prior layer is used to predict new representations for the vertices. There will be a total of K layers with specific graphs, layers will share weight matrices. The more layers, the larger scope the analysis will be in terms of geographical area. In addition to the MPNN representation of the neural network, Long-Short

Term Memory Network will be used to identify the spreading tendencies based on vertex information representation.

Now that the graph representation has been initialized, we can look the overview of the paper experiment that was conducted. Four countries were analyzed - England, France, Italy and Spain. These countries had overlapping time frames as some countries like Italy had a large initial spike compared to those who were hit later like Spain, England and France. In order to track users, the researchers used Facebook's program "For the Good" to track users via smartphone geolocation. This data set constitutes a random

$$X \sim \mathcal{N}(\mu, \sigma^2)\,.$$

and non biased set as the choice is up to the user to detect location, and movement between people will be considered random. Secondly, the authors state that the infection data will come directly from county/regionalized reporting from government health officials and therefore will be considered legitimate official reports. Finally, the assumption is made that a person traveling from one region to another while infected with Covid-19 will lead to an increase in transmission in that destination region.

1.1. Experiment

The authors did a variety of experiments using different combination models; they employed transfer learning (TL) - (intended purpose was to incorporate past knowledge of transmission rates in other countries into their own MPNN. One interesting way an experiment was ran was by training the data on very short term data sets and longer running averages (e.g., 1 day - 14 days). Hyper-parameters were set at 500 epochs with early stopping after 50 epochs of patience which will take place no earlier than 100 epochs. Learning rate was set at 10^{-3}, hidden units in each layer is normalized with dropout of $\frac{1}{2}$. The following neighborhood aggregation scheme is used to represent the updates for the vertices of each of the input graphs:

$$\boldsymbol{H}^{i+1} = f(\mathbf{A}\mathbf{H}^{i}\mathbf{W}^{i+1})$$

where f is a nonlinear activation function (such as ReLU), $\mathbf{W}$ is the corresponding weight matrices for each of the nodes, $\mathbf{H}^i$ is the node representations of the previous layer, with $\mathbf{H}^0 = X$, and $\mathbf{A}$ is the normalized adjacency matrix such that the sum of all weights of the incoming edges of each node is 1.

The authors have noted that they have used skip connections from each layer to the output layer which consists of a sequence of fully-connected layers. Noted is that they also apply a ReLu function to the output of the network as the number of new cases (outputted by the neural network) will always be a natural number. Furthermore, the loss function chosen was MSE.

1.2. Results

Using a time period of 14 days, the authors saw MPNN + TL to be the best performing model that limited error with respect to AVG-WINDOW - floating average number of cases for a specific geographical region. One very interesting point made is that MPNN model keeps up very well with its MPNN + TL counterpart in the short term analysis, but fails in the mid to long range analysis as the TL picks up on more consistent trends. The reason that the MPNN+TL performs exceptionally well over the course of a longer period of time is that the model will pick up on the fact that the symptoms associated with covid-19 can take a bit to appear, meaning that over the course of a 14-day sliding window, MPNN+TL will be able to pick up on those trends much easier.

Next, it is important to touch upon why other models such as the ARIMA, PROPHET and MPNN+LSTM models did not perform as well as the MPNN+TL. Firstly, there is always an inherent difficulty in learning with time-series forecasting, and secondly, the quite erratic epidemic curve led to an additional difficult learning environment for the time series.

1.3. Review

To begin with the critique of this paper, I believe first and foremost that the paper lacks certain variability. While the methods mentioned in the paper take into account past current infection rates and uses that as vertex nodes, the paper does address that it does not take into account population distribution (gender, underlying conditions, etc), weather patterns, and population density among others. While I realize that these are important to the transmission of Covid-19, it is important to note that it would have made this research severely complex and much harder to understand. One of the other critiques I have of this GNN application is that the edges are directed. This means that there is only one edge from locations u to v indicating a means of travel from u to v. This intuitively is a bit odd as one would be able to travel bidirectionally between both nodes. Furthermore, the algorithm states that it does not take into account motion within a location but the figure on page three of the paper shows a self loop on node u. I don't see the purpose of establishing that self loop if it has no implication on the model's analysis of localized movements (which should be irrelevant according to the authors). I do enjoy that the authors provided a future scope of the project in which they addressed the short comings of their model and what where their desires lay for improvement. This work was done in the second quarter of 2020, so they were attempting to refine the models in order to predict when a second wave will hit and how much of an impact it will have.

2. Weight Normalization: A Simple Reparameterization to Accelerate Training of Deep Neural Networks [2]

The second research paper that will be a good addition to GNN modeling of Covid infection rates has to do with reparameterization of the weight vectors between layers in order to condition the optimization of neural networks. According to the authors Tim Salimans and Diederik P. Kingma, the reparameterization is based on batch normalization. This methodology can work hand-in-hand with LSTM's which are spoken with a great deal in the previous paper. The long-short term memory is utilized to a great extent in the Covid GNN model.

Deep learning has shown that first order gradient optimization can lead to low error in model outputs but it is also highly dependent on the curvature of the of the model being optimized. This may be seen as a drawback, but reparameterization such as reducing the cost-gradient of the neural network can improve the neural architecture gradient. A main point in conditioning the neural network gradient amounts is by whitening the input of each layer with an inverse Fisher Information Matrix [3]. On the other hand, and the key contribution of this paper, is the proposal of weight normalization shares common properties of batch normalization without all the noise that is added to the corresponding gradients. Furthermore, no additional overhead is added to the computational cost of each epoch meaning that memory is conserved and additional time complexity is negligible. Therefore making the author's version of reparameterization the preferred method of optimization (with proper initialization).

Let us evaluate a standard neuron that consists of a sum of input features:

$$y = \phi(\boldsymbol{w} \cdot \boldsymbol{x} + b)$$

In this case, $\boldsymbol{w}$ is a weight vector to the corresponding k^{th} dimension, b is a bias scalar, $\boldsymbol{x}$ is the corresponding input feature matrix of the k^{th} dimension. In a traditional sense, standard neural networks are trained via stochastic gradient descent; although, the claim of this paper is that we reparameterize each weight vector with respect to a parameter vector v and a scalar b.

$$\boldsymbol{w} = \frac{g}{\|\mathbf{v}\|}\boldsymbol{v}$$

This has been done before but only after each step of stochastic gradient descent. What sets this apart is that the descent is directly performed with the new parameterized $\boldsymbol{w}$, and by doing so, will improve the convergence of optimization - this is due to taking the vector over its own euclidean norm: $\frac{\boldsymbol{v}}{\|\mathbf{v}\|}$.

2.1. Review

This paper, unlike the previous one sticks to the textbook basics of the structure of neural networks so, for one, does not require large amount of experience with deep neural networks nor does it require the reader to be familiar/up to date with other published work.

That being said, there are some questions I have after going through this paper. It is said that the optimization performs the best when the underlying batch normalization has its parameters properly initialized such that it fixes the scale of features generated by each layer of the neural architecture. On the other hand, the downside of the this is that this initialization method will only work under similar circumstances that batch normalization is applicable. For other models such as in GNN's and LSTM's it will be slightly different to implement but will act the same way on linear and convoluted layers (also with corresponding activation functions). The only slight difference is that in the graphical neural networks that there will be respective embedding updates and message passing. This means that there will be additional overhead when running through the gradient descent of the features without a parameter initialization. This is briefly touched upon but the additional computational time and memory constraint impact is not directly addressed with the use of traditional initialization methods. One question I would like to ask the authors is how the directional weight vector $\boldsymbol{v}$ is chosen. $\boldsymbol{v}$ is described as a k-dimensional vector, but they do not go into depth or mention how it is initialized. On a different note, I would like to ask why the substitution of the scalar g for e^s where s is a log-based parameter to improve the performance of stochastic gradient descent. I understand this is due to the fact that the order of operations and subsequent chain rule in differentiation during back propagation is influenced by magnitude, but I am unsure as to how specific the log scale is chosen.

This paper is very strong in its background information and mathematical support, as well as its explanations as to why a certain parameterization will lead to an increase in optimality. On the other hand, it leaves some unanswered questions regarding the actual importance of specific parameters, rather the authors speak generally on the performance of substitution, ititialization and reparameterization.

3. Mean-field theory of graph neural networks in graph partitioning

This paper provides an excellent theoretical analysis of the performance of GNN's, specifically touches upon GNN ability to act as a classifier that has a lot of flexibility. My one concern immediately with this paper is that it does not provide an applicable definition of ”flexible” so it leads me to believe that it is flexible in which how the features are able to be processed. Subsequently, the paper goes on to analyze whether there is a trade off betweeen the flexibility of GNN's and the high accuracy of its output models. Moreover, there is an analysis on whether or not the high accuracy is directly related to the back propagation or the architectural structure of the neural network itself.

The authors raise a great question which leads the reader to think whether or not the nonlinear structure of GNN's lead to good performances or that the performance is due to the continual learning of model parameters through back propagation. Furthermore, they ask whether or not the architecture is truly influential in performance or would any arbitrary choice suffice. Finally, they touch upon whether or not GNN's truly outperform other models. These questions are answered through the mean-field theory in which the authors perform graph partitioning of a GNN. They will set a architectural structure to the GNN that is essential with random model parameters - in doing so, we see if the architecture is truly indicative of performance. The authors then flip the conditions and fine tune the model parameters and run the same analysis.

It is also important to note that the authors utilized belief propagation (BP), also called message passing algorithm, which is the same algorithm used in the first paper. The message passing is defined by putting directions to every undirected edge on the graph and then determining the marginal probability that a certain vertex i belongs to a group σ. The programmer can then chose the matrix and activation functions that suit the back propagation which will then appropriately update the parameters.

The experiments that were conducted went as follows. They evaluated the performance of an untrained GNN which had the resulting state read using a k-means classifier. In this experiment, zero parameter learning took place. The results that were found was that the numerical experiment without the parameter learning and the trained classifier already performed relatively well under the circumstances. The other experiment involved a trained GNN with back propagation and a trained classifier. The model parameters in this were optimized using a default Adam configuration. The results show that this experiment showed greater performance and more flexibility and generally is considered to have outperformed its rival experiment as well as other models. This is singularly attributed to the back propagation.

3.1. Review

Out of all three of the papers in this report, this one was by far the most difficult to understand. There is extensive math and long paragraphical equations that seem to drag and confuse the reader a bit. This paper is clearly meant for someone with years of thorough experience with neural networks and mean-field theory. Having said that, underlying concepts were able to be grasped. I feel that the paper did a very good job at explaining the two separate experiments and how they would isolate the two claims and perform the analyses. Furthermore, the authors were clearly able to relate message passing algorithm to GNN's which was a great tie in to the first two papers. Moreover, the authors did a good job in explaining in what fashion a certain GNN set up would not perform as well as another one would.

On the other hand, some critiques I have for this paper is that they implemented an unnecessary k-means classifier with the feedforward process of the untrained GNN. The inference of the stochastic model can be performed just as well without the use of any training of a classifier. This naturally leads to the question as to why they chose to include a training procedure to the model. Another question I have is about the section regarding cross-entropy error function. They state that a set of vertices that contain the same labels, that label itself doesn't mean anything - also known as the identifiability problem. This leads to the question as to why the classifier is unable to be trained under these circumstances?

4. Weighted Normalization & Reparameterization Implementation

4.1. Implementation

Let us touch upon the desires to implement my own version of this research paper that is covered in section 2 of this journal. My main desire with weighted normalization of layer weights in a neural network resides on the fact of, not only how simple it is to optimize a deep neural network, but also how versatile it is. The main goal of this implementation was to demonstrate how versatile and simple it is to incorporate reparameterization into the pre-hook forward pass of each stochiastic gradient descent.

Let us first touch upon the dataset that was used in the training and testing of the model that will be discussed shortly. I used the CIFAR-10 colored image dataset. The CIFAR10 dataset has the classes: 'airplane', 'automobile', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'truck'. The images in CIFAR-10 are of size 3x32x32, i.e. 3-channel color images of 32x32 pixels in size. I will preface this by explaining that the two convoluted neural networks are slightly different, as weighted normalization is more optimized under certain architectures. Both neural networks start around the same loss, and finish at the same accuracy (roughly around seventy percent), so that is the reasoning behind why we can use similar models that aren't exact to perform an analysis on this reparameterization. With this being stated, the deep neural network that contained the reparameterization consisted of two convoluted layers followed by two linear layers. The convoluted layers expanded from 3 $\longrightarrow$ 16 $\longrightarrow$ 48 channels with a kernel size of 5 and a stride of 1. The weight normalization was applied to the first convoluted layer and the first linear layer; I found this to be the most efficient implementation of a basic neural network structure. Each convoluted layer is separated by a max pooling of size (2x2). Linear rectification activation functions were applied to each convoluted layer and the first linear layer. In order to go about the evaluation of the weighted normalization, I also checked the difference between two combinations of activation and loss functions: Option 1: Use log_softmax() activation on the output nodes in tandem with F.nll_loss() within training epochs ("negative log-likelihood loss"). Option 2: a. No activation application the output nodes (or equivalently, identity() activation) or b. a sigmoid activation on the output, in conjunction with cross entropy loss when training. While essentially somewhat similar, the results I found were that the negative log-likelihood loss was slightly more applicable when performing prediction on test labels (binary classification). This is due to the fact that the softmax returns two values that sum to one in order to determine the class of the label. The crucial difference between the two cost functions: the negative log-likelihood considers strictly the output for the corresponding class, whereas the cross-entropy function also considers the other outputs as well. Due to this, I decided the strict nature of softmax was applicable to our classification.

In order for the weight normalization to work, and very similar to the necessity of a Forward() function within a class that supers to nn.Module, we need a WeightNorm() class that contains various necessary callable functions such as compute_weight(), static apply(), remove(), __call__, remove_weight_norm(). The function calls are all necessary in order to perform prehooking which occurs before every forward pass. A hook is just a callable object with a predefined signature, which can be registered to any nn.Module object. When the trigger method is used on the module, such as the forward() call, the module itself is passed to the hook, executing before the computation proceeds to the next module (PyTorch, 2019). The function apply will receive the nn.Module before the forward pass and will check if a WeightNorm instance of that node has been computed before, if so, raise a runtime exception, else reparameterize the node. The function will proceed to decouple the "weight" attribute from the module into both the direction and magnitude. The decoupling occurs by calling an overriden function named "norm_except_dim" (PyTorch, 2019). This function will contract the norm along the dimension and power provided, very similar to regular norm, but regular norm will contract along all other dimensions as well. The call to this function returns a normalized tensor 'v' that acts as the directional portion of the decoupled weight attribute. For this set of functions to work we need to use norm_except_dim, but otherwise it is an obsolete function within PyTorch library as a regular norm(tensor) call will do the same thing. Finally, the magnitudal vector 'g', that was the original weight vector, is now stored as a tensor of corresponding scalars that when multiplied with the 'v' tensor and the norm of 'v'. This is then returned to the apply function and reset as the new 'weight' attribute of the model.

In the opposite, corresponding network, I had constructed a nonweighted neural network that the weighted one was evaluated against. This network had a test accuracy of roughly 70 percent which is what the weighted average was at. The structure consisted of 3 convoluted layers (3 $\longrightarrow$ 16 $\longrightarrow$ 32 with a kernel size of three and a padding one 1), separated by max pooling layers that were (2,2). Then transitioned to three fully connected layers separated by dropouts of $\frac{1}{2}$.

The optimizer for the training is stochastic gradient descent with a learning rate of 0.01 and a momentum of 0.8. This is generally a very safe optimizer to use for efficient fitting with regards to linear classifiers and convex loss function such as the negative log-likelood function that we are using for testing. This way our optimizer and loss function are compatible. A learning rate of 0.01 and momentum of 0.8 are very standard across the industry for general neural network classification; of course I can optimize these parameters and see which one works the best, but under the scope of this project, which is to show the from-scratch implementation of weight normalization, I did not find this necessary.

Moving on to the next implementation evaluation, I decided to see how the weighted reparameterization stacked up against a non weighted normalization in a generic auto-encoder to remove noise from MNIST generated data. Similar to the CIFAR-10 data, MNIST is a 28x28 pixel wide image but one major difference is that the number of channels is only one as it is a grey scale picture and not RGB like CIFAR-10. The auto-encoder is set up similar to the convoluted neural network implemented earlier except that it is strictly convoluted with each sequential forward() operation separated by a linear rectification activation function. The first convoluted encoding layer has its weight parameter normalized as well as the first transposed convoluted decoding layer. The output of the decoding layer has a sigmoid function applied to constrict the output module tensor to contain values between 0-1. This allows us to have a value associated with the grey scale pixelation at that certain location, allowing for our loss_fn() (MSE_loss function). Ideally, since I apply a sigmoid function on the output, MSE loss will be more beneficial in training the model as it will punish the model more severely for larger errors in the classification, whereas something like negative log likelihood that was used in first CIFAR-10 classification was not constricted by a sigmoid function, rather flatted linearly as an output. Since both of the weighted and non-weighted neural networks are constricted by a sigmoid, and use MSE loss, we can use the same neural network structure as the pairing of the loss function and constricting leads to very similar outputs of the network - unlike the flattening and the negative log likelood of the CIFAR-10 classification.

4.2. Results

The evaluation method for this section is based on the loss for each model and ending test accuracy of the models. The lower the model's loss is, the better the model is, lower loss means higher test accuracy. Both the image classification and the autoencoder models are run using my custom implementation of weight normalization as well as PyTorch's own implementation. Results are shown below in two sets of graphics.

This section discusses the head-to-head matchup between weighted and non weighted neural networks. The results are relatively straight forward: the red linear line corresponds to the weighted normalization of the neural network and the blue represents the nonweighted neural network of the same type. The following graphics show the performance of the CIFAR-10 classification using the aforementioned neural network architecture.

Figure (a) Self Implementation

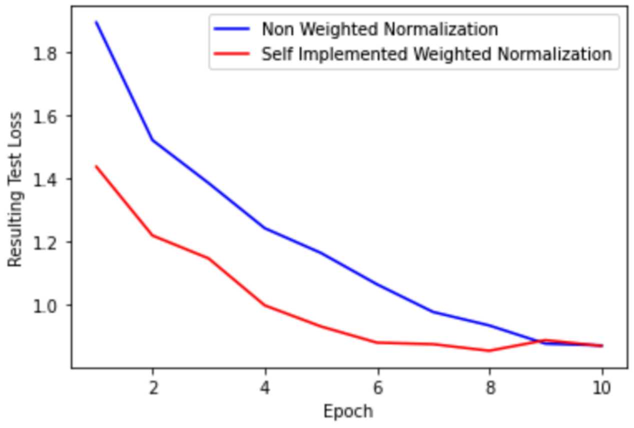
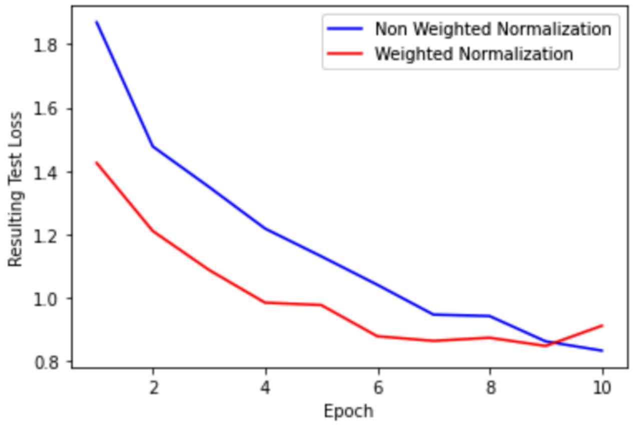


Figure (b) PyTorch Implementation



As we see, both the red lines (weight normalization) of our neural network leads to increased optimization of test loss for each epoch. Both networks end around a seventy percent test accuracy which is quite good for an RGB image classification with a basic deep neural network. One note here, the PyTorch default implementation does spike toward the end which indicates that over-training has occurred and possibly early stopping would have been beneficial. The following images are of the same style, the generic autoencoder to remove noise from the images is in blue and the weighted normalization is in red. Very similar to the previous results, my implementation both outperformed PyTorch implementatoin initially but PyTorch implementation seemed to have a steeper descent and finished with a higher test accuracy than mine. This is, of course, if we take into account the spike at the end of the previous graphs and attribute it to outlier overtraining.

Figure (a) Self Implementation

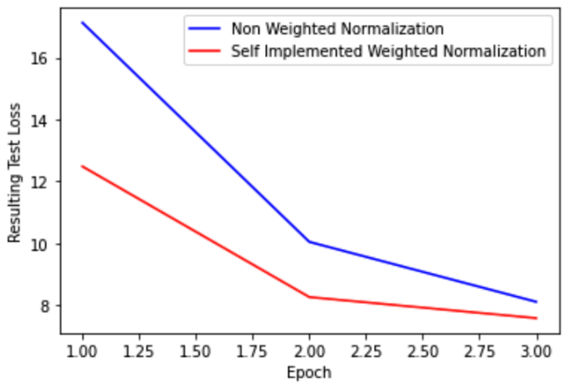
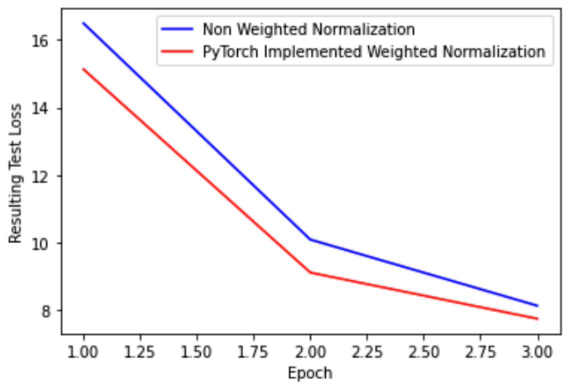


Figure (b) PyTorch Implementation



As we see, there is a certain beauty and versatility to weight normalization, and paired with how simple it is to implement, it should be an easy choice for those looking to optimize their models. Of course it isn't always the ideal choice as there is an overhead associated with the implementation. Clearly if we have a large enough data set or the data is data-dense, we will have a trade off between overhead and performance.

4.3. Discussion

These results are relatively straight forward to analyze, but I think that the underlying reason that weight normalization is effective is very interesting. It is important to note, as previously mentioned, that my implementation of weight normalization started off with a lower loss value but failed to finish as strong as non-weighted normalization and PyTorch's own implementation of such. This leads to the question of whether or not weighted normalization is much more susceptible to overtraining. Furthermore, the discussion arises on the subject of would it be more beneficial to actively monitor the models performance and have an early stopping? I believe the answer to that is yes. In a case where we have immediate or initial performance increase, we may be increasingly susceptible to an overtrained model in sequential epochs following that level off of model performance. This is clearly indicated in the PyTorch implementation, but is also implicitly shown in my implementation as mine levels off quite a bit in mid to later stage epochs.

Moving forward, the topic to address is why does my implementation start with a much lower loss value vs. PyTorch implementation. As I do not have access to PyTorch's _weight_loss function directly, one can only suspect why this is the issue. One idea, when observing the PyTorch function call [2] is that PyTorch takes the norm over a optimized power and also takes into account if a dimension is passed into the function that is not ideal, it will return a defined dimension that is appropriate to take the norm over. Also, I would suspect that my method is computational heavy on the front end, meaning that, against PyTorch implementation, I rely on the immediately normalization_except_dimension over a set power of two whereas their implementation accounts for possible changes in powers.

Continuing on, we can try to analyze why both the weighted normalization models get outperformed in later epochs: I am led to believe that this shortcoming has to do with the initial impulse of training optimization and, when we pass a critical point, there becomes a linear decay in the derivative of our loss reduction. This may be due to the fact that reparameterizing weights only has initial impact on the initial corresponding epochs, respective to data input. This is supported by the initial paper in which the authors state that (referring to the combined mean-only batch normalization + weight normalization) that the combined normalization represents the best performance for CIFAR-10 among methods that do not use data augmentation [4]. The reason they mention this is because the weight normalization provides the best initial accuracy with lower tail end performance, but this is offset with the mean-only batch normalization that is tailored to the approximate Gaussian noise that is present in the pictures.

Concluding with the analysis of the implementation, I would like to touch upon the importance of weighted normalization. Especially with stochastic gradient descent optimization, weighted normalization works great, and even better with other reparameterization methods such as mean-only batch normalization that is able to pick up on the faults that weighted normalization sees when it is implemented by itself. The usage of weighted normalization has been steadily increasing, and some have made it a mandatory default within deep learning architectures due to its ease of use, low computational overhead, and the ability to pair it with other reparameterization techniques.

4.4. Future Implementation

As someone who finds this subject area extremely interesting, I would very much like to apply this to outside research papers, such as Covid-19 modeling with GNN's that is touched upon in section 1. This would be an excellent way to pair this research paper together, and could possibly even increase the optimizability of the network. In addition to this, my future desires lay in re-implementing the combination of reparameterization techniques touched upon in this paper. The reason I did not re-implement this paper was due to the fact that I wanted to strictly show the purpose and benefit of strictly implementing weighted normalization, but now that this is shown, I can move forward an use a combination of techniques.

References

George Panagopoulos, Giannis Nikolentzos, M. V. Transfer graph neural networks for pandemic forecasting. Neural Information Processing Systems, 2020.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., and Chintala, S. Pytorch: An imperative style, high-performance deep learning library. In Wallach, H., Larochelle, H., Beygelzimer, A., d'Alché-Buc, F., Fox, E., and Garnett, R. (eds.), *Advances in Neural Information Processing Systems 32*, pp. 8024–8035. Curran Associates, Inc., 2019.

PyTorch. Weighted$_norm,$ 2019.

Tatsuro Kawamoto, Masashi Tsubaki, T. O. Mean-field theory of graph neural networks in graph partitioning. 2-3-26 Aomi, Koto-ku, Tokyo, Japan; 2-12-1 Ookayama Meguro-ku Tokyo, Japan, 2018. ”Neural Information Processing Systems”.

Tim Salimans, D. P. K. Weight normalization: A simple reparameterization to accelerate training of deep neural networks. Neural Information Processing Systems, 2016.