


Loss-contribution-based in situ visualization for neural network training

T.-Y. Lee[†]

¹Information Technology RD Center, Mitsubishi Electric Corporation, Kamakura, Kanagawa, Japan

Abstract

This paper presents an in situ visualization algorithm for neural network training. As each training data item leads to multiple hidden variables when being forward-propagated through a neural network, our algorithm first estimates how much each hidden variable contributes to the training loss. Based on linear approximation, we can approximate the contribution mainly based on the forward-propagated value and the backward-propagated derivative per hidden variable, both of which are available during the training with no cost. By aggregating the loss contribution of hidden variables per data item, we can detect difficult data items that contribute most to the loss, which can be ambiguous or even incorrectly labeled. For convolution neural networks (CNN) with images as inputs, we extend the estimation of loss contribution to measure how different image areas impact the loss, which can be visualized over time to see how a CNN evolves to handle ambiguous images.

CCS Concepts

• **Human-centered computing** → Scientific visualization;

1. Introduction

In the high performance computing (HPC) area, HPC applications, typically scientific simulations, can take days or weeks to run, and in situ visualization [CAA*20] aims doing visualization with simulations together so once a simulation starts, users can immediately visualize the intermediate result without waiting. As the training of neural networks is also a time-consuming process, it will be ideal if the training process can be extensively visualized in situ as well.

While several methods have been presented to visualize the internal parts of neural networks, it is difficult to apply these methods in situ. As a neural network is essentially a cascade of multiple operators, to understand how a neural network processes a data sample or called *data item* hereafter, it is required to analyze the intermediate results by the operators, which in total can generate thousands or millions of values. These intermediate results are storage-consuming and difficult to exhaustively examine. Existing visualization techniques [WGSY19, LSL*17, WGYS18, ZBOT19] require pre-processing to extract and aggregate related information, which can be time-consuming. Consequently, running these algorithms in situ can slow down the training and is thus undesired. Also, existing visualization techniques might be dedicated to a specific task like image classification [SCD*17, ZKL*16, SVZ14, ZF14], which is hard to apply to other tasks.

In this paper, we present an in situ visualization algorithm to

monitor the training process of neural networks. The key idea is to quantitatively estimate that given a data item, how much its intermediate results within a neural network, or hidden variables, contribute to the training loss. As the loss is essentially the difference between the neural network result and the expected output, if a hidden variable has high loss contribution, it means that due to this variable, the result cannot match the expected output. Based on data items, operators, and parameters that are related to these highly contributing hidden variables, we can form more effective visualization.

Since in situ algorithms should not cause too much overhead to the training process, we aim to re-use existing intermediate results during the training process as much as possible. For this purpose, we model the relationship between the calculated hidden variables per operator and the final training loss as a linear model. By extending the linear assumption, we can approximate the relationship mainly based on the forward-propagated value and the backward-propagated derivative per hidden variable, both of which are already computed by the training process.

Based on the loss contribution in the level of hidden variables, we can estimate the loss contribution in the level of data items by aggregating the contribution of corresponding hidden variables. This allows us to select difficult data items that contribute most to the training loss. Furthermore, if the data are images and the neural networks involve convolution operators, meaning that each hidden variable can correspond to a specific image area, we can estimate how much each image area contributes to the loss.

[†] Lee.Teng-Yok@ap.MitsubishiElectric.co.jp

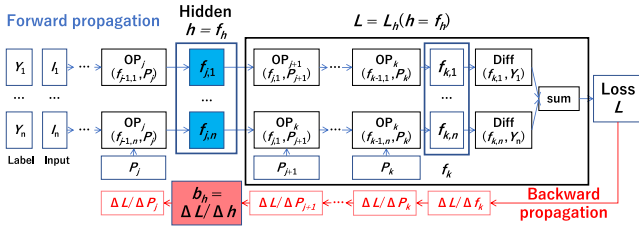


Figure 1: Modeling the relationship from a hidden variable h to the final loss L as a function $L = L_h(h)$.

To verify our algorithm, we extend a public tensorflow-based library *TF-Slim* [Ser16] with our algorithm and tested various image data sets. Our tests show that our algorithm can detect difficult data items to train, which could be very ambiguous to process or even with incorrect labels. Also, we can monitor how the neural network is evolved to enhance its processing ability on ambiguous cases. We also analyze the computation overhead and discuss further directions to optimize.

2. Method

Background We first review terminologies involved in the training of neural networks, which is also illustrated in Figure 1. Given a batch of data items I_i , each of which has an expected result Y_i or called a *label*, we can use them to train a neural network by alternating between *forward propagation* (FP) and *backward propagation* (BP). Here we assume that the neural network is a cascade of K operators $OP_1 \dots OP_K$. Each OP_j has its parameters P_j , which will be updated during the training. FP computes each item I_i through the operators, and each OP_j generates an intermediate result called *hidden variables* $h_{j,i}$, which will be the input for the next operator. Once the output $h_{k,i}$ of the last operator OP_k is computed, the training process computes the *loss*, essentially the sum of differences between the value of $h_{k,i}$ and Y_i of all items. Then it applies BP to compute the derivatives $\frac{\Delta L}{\Delta P_j}$ of all parameters P_j , followed by gradient descent to update the parameter values.

Hidden variables During this process, hidden variables play several important roles. Given the j -th operation, hereafter we denote the concatenated form of hidden variables $h_{j,i}$ of all items as h . First, as hidden variables h are the output of j -th operator OP_j , BP needs to compute the derivative $\frac{\Delta L}{\Delta h}$ first so it can apply the chain rule to compute the derivative $\frac{\Delta L}{\Delta P_j} = \frac{\Delta L}{\Delta h} \frac{\Delta h}{\Delta P_j}$. Second, hidden variables also represent the intermediate state of each input data item. If the hidden variables of any data items contain unusual patterns, it might indicate sub-optimal or even incorrect processing within the neural network, and consequently, the neural network fails to generate the expected output. Thus the goal of our in situ visualization is to detect and display data items that cause high training loss on the fly. We also want to indicate the relevant parts in their input and related operators in the neural network that make these items difficult.

Loss contribution of hidden variables To achieve our goal, our approach models the relationship between the hidden variable h of

each operator and the training loss L as a linear function. Given a hidden variable h , we denote its forward-propagated result as f_h , and consider the operators after h till the computation of training loss L as a function $L = L_h(f_h)$, which are enclosed by the black box in Figure 1. While h is typically a multi-dimensional tensor, without loss of generality, here we consider h as a vector of m elements, and we aim to detect which elements of h contribute most to the outcome L_h .

Loss contribution estimation Our approach estimates the contribution by approximating the function L_h as a linear function in Equ. 1:

$$L_h(h = f_h) \propto \langle f_h, w_h \rangle = \sum_{i=1}^m f_h[i] \times w_h[i] \quad (1)$$

Here w_h is a vector of the same length of f_h where element $w_h[i]$ represents the weight for the i -th element of h . If such a linear relationship exists, how much each element i of h contributes to the sum is related to $f_h[i] \times w_h[i]$.

Contribution approximation To linearly approximate $L_h(h = f_h)$, we can first apply Taylor expansion to expand L_h . Equ. 2 shows the Taylor expansion of L_h with respect to a vector $\mathbf{0}_h$, meaning m elements with all zeros:

$$L_h(h = f_h) = L_h(\mathbf{0}_h) + \langle f_h, \frac{\Delta L_h(h = \mathbf{0}_h)}{\Delta h} \rangle + \dots \quad (2)$$

By dropping the constant term $L_h(\mathbf{0}_h)$ and the higher order terms, we can obtain a form that is similar to Equ. 1. Nevertheless, this will involve an extra calculation of $\frac{\Delta L_h(h = \mathbf{0}_h)}{\Delta h}$. As each layer h has its $\mathbf{0}_h$, computing $\frac{\Delta L_h(h = \mathbf{0}_h)}{\Delta h}$ for all layers can increase the training time and is thus undesired.

In situ approximation Our approach resolves this issue by further approximating Equ. 2 with the following assumption: if the function $L_h(h)$ can be linearly approximated between $h = \mathbf{0}_h$ and $h = f_h$, we assume that the function $L_h(h)$ between h and L_h can be modeled as a hyperplane. Since the derivative on a hyperplane is identical everywhere, $\frac{\Delta L_h(h = \mathbf{0}_h)}{\Delta h}$ should be close to $\frac{\Delta L_h(h = f_h)}{\Delta h}$. Thus we can replace $\frac{\Delta L_h(h = \mathbf{0}_h)}{\Delta h}$ in Equ. 2 by $\frac{\Delta L_h(h = f_h)}{\Delta h}$, which leads to Equ. 3. By denoting $\frac{\Delta L_h(h = f_h)}{\Delta h}$ as b_h , meaning the backward propagated derivative at h , we can simplify Equ. 3 to Equ. 4, our equation to estimate the loss contribution in situ:

$$L_h(h = f_h) \propto \langle f_h, \frac{\Delta L_h(h = \mathbf{0}_h)}{\Delta h} \rangle \approx \langle f_h, \frac{\Delta L_h(h = f_h)}{\Delta h} \rangle \quad (3)$$

$$= \langle f_h, b_h \rangle \quad (4)$$

Data-wise aggregation Based on the element-wise loss contribution in Equ. 4, we can further estimate the loss contribution per input data item. Given the i -th data item with hidden variable h_i in the layer h , we can estimate its loss contribution by aggregating the contribution of all elements of h_i . A straightforward approach is to sum the contribution together, as shown in Equ. 5:

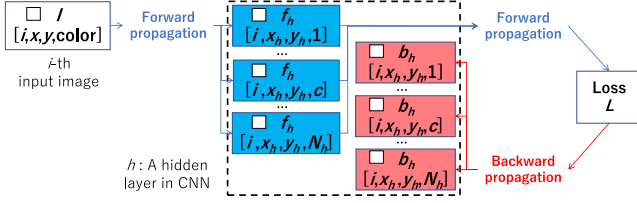


Figure 2: Hidden variables of the i -th image within a CNN. Here we split f_h and b_h based on the channel dimension c . In f_h and b_h , points with same 2D coordinates (x_h, y_h) of all channels c represent the same area in the input image, as illustrate by the white boxes.

$$\sum_{j \in h_i} f_h[j] \times b_h[j] \quad (5)$$

Convolution neural networks Our estimation of loss contribution can be applied to convolution neural networks (CNN) [LBD*89], which are neural networks with convolution operators and commonly used to process images. When training a CNN, the batch of input images is often modeled as a 4D tensor $I[i, x, y, c]$ where i , x , y , and c represent the image, horizontal coordinate, vertical coordinate, and color, respectively, and the hidden variable h will be a 4D tensor $h[i, x_h, y_h, c]$ too. Here the last dimension c is also called a *channel*, and each layer h can have its number of channels N_h . Figure 2 illustrates the hidden variables of a layer within CNN. The forward-propagated $f_h[i, x_h, y_h, c]$ per channel c is also a 2D map, which is also called a *feature map*. The corresponding backward-propagated $b_h[i, x_h, y_h, c]$ has the same shape as f_h . The same 2D location (x_h, y_h) of $f_h[i, x_h, y_h, c]$ and $b_h[i, x_h, y_h, c]$ of all channels c correspond to the same area in the i -th input image, which is illustrated by the white boxes in f and b blocks and the input image I .

Feature maps in CNN An important property of CNN is that each feature map of the hidden variables represents the result of a sequence of convolution operators over the corresponded image. As a result, visualizing the feature maps can indicate how the CNN processes each area [LSL*17, WGYS18, ZBOT19, ZKL*16]. As in situ algorithm cannot afford to dump and show all feature maps, we aim to efficiently aggregate the feature maps into representative ones.

Loss contribution and reduction Our current approach aggregates the feature maps into two maps. Given a layer h with spatial dimensions (x_h, y_h) , one map indicates how much each area in the feature maps *contributes* to the loss by aggregating only *positive* loss contribution of all channels, as shown in Equ 6:

$$C_h(x_h, y_h) = \sum_{c=1}^{N_h} \max(f_h[i, x_h, y_h, c] \times b_h[i, x_h, y_h, c], 0) \quad (6)$$

The other map indicates how much each area in the feature maps *reduces* the loss by aggregating only *negative* loss contribution of all channels, as shown in Equ 7:

$$R_h(x_h, y_h) = - \sum_{c=1}^{N_h} \min(f_h[i, x_h, y_h, c] \times b_h[i, x_h, y_h, c], 0) \quad (7)$$

Hereafter the two maps C_h and R_h are called *loss contribution map* and *loss reduction map*, respectively. Section 4.2 and the supplementary video describe examples to use both C_h and R_h to see how a CNN is evolved during the training.

3. Implementation

We implemented a prototype with our in situ algorithm by extending the tensorflow-based library *TF-Slim* [Ser16], which provides various pre-defined neural networks and optimization algorithms, and supports several public data sets. Besides TF-Slim's original parameters, our prototype needs the following extra parameters:

H_s : Name of a hidden layer to guide the selection of data items. At each iteration of the training process, after forward- and backward-propagation, we compute the loss contribution at layer H_s per data item and select data items with highest loss contribution.

N_s : Number of data items to select based on the loss contribution of layer H_s . Note that as the training of neural network often splits the training data into multiple batches, N_s is the number of items to select from the entire data set, not just from an individual batch. Our prototype uses a priority queue to maintain at most N_s data items during the training.

H_m : A set of hidden layers to monitor during training. For each of the selected data items, we compute and store the loss contribution at these layers. If a hidden layer h has spatial dimensions x_h and y_h , we also compute its loss contribution map C_h and loss reduction map R_h .

For the visualization part, as TF-Slim already uses TensorBoard [WSW*18] to monitor the training process, our prototype extends TensorBoard by rendering the visualization to images and saving these images into the log files used by TensorBoard. Once the training starts, users can use TensorBoard to open the log files to see our in situ visualization. Section 4 describes our visualization with use cases to demonstrate its effectiveness, and we recommend to check the supplementary video to see more description about our enhanced TensorBoard.

4. Use cases

4.1. Ambiguous training data detection

The first part of our in situ visualization is called *Selection View*, which displays the N_s selected images during the training. Once all batches have been trained once, which is called an *epoch*, our prototype displays the selected images as an array of icons to the TensorBoard.

In this experiment, we trained an image classifier CNN called *cifarnet* [Kri09], as illustrated in Figure 3 (a). Given an input image, an image classifier estimates the probabilities of pre-defined object classes. We trained cifarnet with a data set CIFAR10 [Kri09], which was proposed with cifarnet and contains 60000 color icons of 32×32 pixels of ten object classes. In cifarnet, we used layer *Logit* as H_s and layers *pool1* and *pool2* as H_m , which are marked as blue cells in Figure 3 (a). We used the same training parameters provided by the sample script to train CIFAR10 in TF-Slim [Ser16].

Figure 3 (b) shows the Selection View after 10 epochs, which

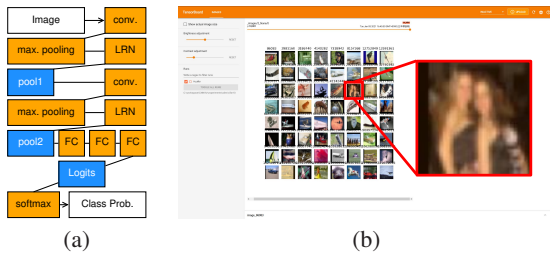


Figure 3: The Selection View (b) when training the CNN cifar-net [Kri09] (a) on the CIFAR10 data set [Kri09]. The enlarged icon shows one of the selected images. While its label is trucks, this image actually contains no trucks. Instead, it contains two persons.

displays 64 selected images of CIFAR10. Among these images, we surprisingly found a potentially incorrectly labeled image. The image is highlighted in the right of Figure 3 (b), which looks like two persons. CIFAR10, however, does not have labels for persons. Instead, this image was labeled as *trucks*. In such a case, it will be difficult (and actually meaningless) to train the CNN to classify the two persons as trucks.

This result clearly demonstrates the benefit of our in situ visualization. Although CIFAR10 is widely used nowadays, to our knowledge, this is the first time that this mis-labeled image is identified. One possible reason is that CIFAR10 contains 60000 icons, which is difficult to be individually examined. With our algorithm, users can identify issues like incorrect labeling without waiting for the training to finish.

4.2. Enhanced monitoring of training

To understand how the neural network is evolved, once encountering a previously selected image, our prototype organizes its C_h and R_h into a *Monitoring View*. Figures 4 (b) and (c) shows the Monitoring Views of an image in the MNIST data set [LBBH98]. In the Monitoring View, each column represents a hidden layer h in H_m , and the top and bottom icons show its C_h and R_h , respectively. The two maps use the same color mapping, as shown in the color bar above. Some extra information are shown in the top of the column, including the layer name, the output (O), and the label (L). The rightmost column shows the images before being forward-propagated through the neural network.

In this experiment, we used the data set MNIST [LBBH98] to demonstrate the effectiveness of Monitoring Views. MNIST is a data set for hand-written digit recognition, which contains 60000 icons of 28×28 pixels of hand-written digits. We trained a CNN called LeNet, which was proposed with MNIST, to estimate the probabilities of digits 0 - 9 within an image icon. Figure 4 (a) shows the architecture of LeNet where we used layer *Logit* as H_s and layers *pool1* and *pool2* as H_m , all of which are marked as blue cells. We used the same training parameters provided by the sample script of LeNet in TF-Slim [Ser16].

The image used by Figures 4 (b) and (c) is an ambiguous image in MNIST, which represents digit 3 but was difficult to classify at the beginning. Figure 4 (b) shows the Monitoring View when this image was incorrectly classified as 7. From the top C_h of the

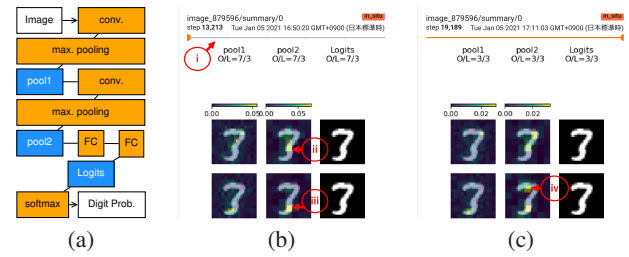


Figure 4: Monitoring Views of an ambiguous digit 3 of the MNIST data set [LBBH98] when training the CNN LeNet (a) [LBBH98]. Users can use the top slider (i) to see the maps C_h and R_h at different iterations. (b): The maps at an earlier iteration when the image was incorrectly classified as 7. (c): The maps at a later iteration. Please see texts for the description about markers (ii)-(iv).

middle column, we can see bright spots in the middle (marked by (ii)), meaning that this area has high loss contribution. In contrast, in the bottom R_h , the lower part of the digit (marked by (iii)) was highlighted, meaning that this part helps to reduce the loss. This is reasonable because digits 3 and 7 have very different lower parts.

In the top of each Monitoring View, there is a horizontal slider (marked by (i)), which can be used to see the maps at different iterations. Figure 4 (c), for instance, shows the Monitoring View of the same image at a later iteration when the neural network began to correctly classify this image. In the bottom R_h of the middle row, we can see a bright spot (marked by (iv)) on the top curve of the digits 3. This implies that in the later stage of the training, this neural network can also emphasize this part, which helps to correctly recognize this digit as 3.

5. Discussion

Performance One benefit of our in situ algorithm is that the calculation is very simple and thus the computation overhead is small. This is especially apparent on large convolution neural networks because convolution is slow. To verify, we tested our algorithm per Equ. 5, Equ. 6 and 7 to the 50 convolution layers of ResNet50 [HZRS15]. With a batch of 16 images of 224×224 pixels, the FLOPS was increased from 337M to 339M, meaning that the overhead is smaller than 1%.

Comparison with other in situ algorithms As in situ processing is a broad topic, there could be different types of in situ algorithms, as reviewed by Child *et al.* [CAA*20]. To categorize our algorithm, the execution is divided by time, as it needs to examine hidden variables after each iteration of training. It is also application-aware because we extended TF-Slim's code that iterates the training process.

Future work In the future, we will apply our in situ algorithm to other tasks like object detection or natural language processing. While different tasks require different types of loss, our algorithm is independent to the loss types. Implementation-wise, we would like to replace TensorBoard by our own user interface so we can optimize the rendering speed and customize the user interaction.

References

- [CAA*20] CHILDS H., AHERN S. D., AHRENS J., BAUER A. C., BENNETT J., BETHEL E. W., BREMER P.-T., BRUGGER E., COTAM J., DORIER M., DUTTA S., FAVRE J. M., FOGAL T., FREY S., GARTH C., GEVECI B., GODOY W. F., HANSEN C. D., HARRISON C., HENTSCHER B., INSLEY J., JOHNSON C. R., KLASKY S., KNOLL A., KRESS J., LARSEN M., LOFSTEAD J., MA K.-L., MALAKAR P., MEREDITH J., MORELAND K., NAVRÁTIL P., O'LEARY P., PARASHAR M., PASCUCCI V., PATCHETT J., PETERKA T., PETRUZZA S., PODHORSZKI N., PUGMIRE D., RASQUIN M., RIZZI S., ROGERS D. H., SANE S., SAUER F., SISNEROS R., SHEN H.-W., USHER W., VICKERY R., VISHWANATH V., WALD I., WANG R., WEBER G. H., WHITLOCK B., WOLF M., YU H., ZIEGELER S. B.: A terminology for in situ visualization and analysis systems. *The International Journal of High Performance Computing Applications* 34, 6 (2020), 676–691. 1, 4
- [HZRS15] HE K., ZHANG X., REN S., SUN J.: Deep residual learning for image recognition. *arXiv:1512.03385* (2015). 4
- [Kri09] KRIZHEVSKY A.: *Learning Multiple Layers of Features from Tiny Images*. Tech. rep., University of Toronto, 2009. 3, 4
- [LBBH98] LECUN Y., BOTTOU L., BENGIO Y., HAFFNER P.: Gradient-based learning applied to document recognition. *Proceedings of the IEEE* 86, 11 (1998), 2278–2324. 4
- [LBD*89] LECUN Y., BOSER B., DENKER J. S., HENDERSON D., HOWARD R. E., HUBBARD W., JACKEL L. D.: Backpropagation applied to handwritten zip code recognition. *Neural Computation* 1, 4 (1989), 541–551. 3
- [LSL*17] LIU M., SHI J., LI Z., LI C., ZHU J., LIU S.: Towards better analysis of deep convolutional neural networks. *IEEE Transactions on Visualization and Computer Graphics* 23, 1 (2017), 91–100. 1, 3
- [SCD*17] SELVARAJU R. R., COGSWELL M., DAS A., VEDANTAM R., PARIKH D., BATRA D.: Grad-cam: Visual explanations from deep networks via gradient-based localization. In *ICCV* (2017), pp. 618–626. 1
- [Ser16] SERGIO GUADARRAMA, NATHAN SILBERMAN: TensorFlow-Slim: A lightweight library for defining, training and evaluating complex models in tensorflow, 2016. URL: <https://github.com/google-research/tf-slim>. 2, 3, 4
- [SVZ14] SIMONYAN K., VEDALDI A., ZISSERMAN A.: Deep inside convolutional networks: Visualising image classification models and saliency maps. In *ICLR* (2014). 1
- [WGSY19] WANG J., GOU L., SHEN H.-W., YANG H.: DQNviz: A visual analytics approach to understand deep Q-networks. *IEEE Transactions on Visualization and Computer Graphics* 25, 1 (2019), 288–298. 1
- [WGYS18] WANG J., GOU L., YANG H., SHEN H.-W.: GANviz: A visual analytics approach to understand the adversarial game. *IEEE Transactions on Visualization and Computer Graphics* 24, 6 (2018), 1905–1917. 1, 3
- [WSW*18] WONGSUPHASAWAT K., SMILKOV D., WEXLER J., WILSON J., MANÉ D., FRITZ D., KRISHNAN D., VIÉGAS F. B., WATTENBERG M.: Visualizing dataflow graphs of deep learning models in tensorflow. *IEEE Transactions on Visualization and Computer Graphics* 24, 1 (2018), 1–12. 3
- [ZBOT19] ZHOU B., BAU D., OLIVA A., TORRALBA A.: Interpreting deep visual representations via network dissection. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 41, 9 (Sept. 2019), 2131–2145. 1, 3
- [ZF14] ZEILER M. D., FERGUS R.: Visualizing and understanding convolutional networks. In *ECCV* (2014), pp. 818–833. 1
- [ZKL*16] ZHOU B., KHOSLA A., LAPEDRIZA A., OLIVA A., TORRALBA A.: Learning deep features for discriminative localization. In *CVPR* (2016), pp. 2921–2929. 1, 3