A MATLAB-based tutorial on implementing custom loops for training a deep neural network

Mahsa Yousefi and Ángeles Martínez Calomardo

Department of Mathematics and Geosciences, University of Trieste, Italy

Abstract
While there are many Python-based codes provided by authors in DL literature and related repositories, there are limited resources for MATLAB users to customize their own optimizer for training a deep neural network (DNN). We have contributed to fill this gap by providing basic intuition on designing and implementing a DNN, and computing required quantities of any prescribed training algorithm for performing a specific task such as image classification. This tutorial concerns basic general notes of implementation using the MATLAB Deep Learning toolbox that has been used in more details in the programming of the algorithms used in [3].

KEYWORDS
MATLAB deep learning toolbox, training loops, nonlinear programming, stochastic quasi-Newton methods, deep neural networks

1. Introduction

In [3], we have analyzed the behaviour of two well-known quasi-Newton methods, named sL-BFGS-TR and sL-SR1-TR, applying an stochastic trust region framework on deep neural network architectures in a deep learning application such as image classification. Since the considered training algorithms are not defined as built-in functions, we have exploited the Deep Learning Custom Training Loops of MATLAB to customize their iterations and implement them for training. Implementation details of these two stochastic QN algorithms are available at: https://github.com/MATHinDL/sL_QN_TR/. In this tutorial, we provide a general implementation notes concerning the MATLAB DL Toolbox by which one can learn how to define an initialized convolutional neural network (CNN) and how to compute function, gradients, and other quantities required per single iteration of any gradient-based optimization algorithms.

2. Implementation

2.1. Constructing a network

In supervised learning, the goal is to minimize the empirical risk of the model (DNN) by finding an optimal parametric mapping function $h(\cdot; w) : \mathbb{R}^n \longrightarrow \mathbb{R}$

$$\min_{w\in\mathbb{R}^n} F(w) \triangleq \frac{1}{N}\sum_{i=1}^{N} L(y_i, h(x_i; w)) \triangleq \frac{1}{N}\sum_{i=1}^{N} L_i(w), \tag{1}$$

where $w \in \mathbb{R}^n$ is the vector of trainable parameters of the model and (x_i, y_i) denotes the ith sample pair in the available training dataset $\{(x_i, y_i)\}_{i=1}^N$ with input x_i and target y_i. Moreover, $L_i(w)$ is a loss function defining the prediction error between the model's output $h(x_i; w)$ and the target y_i converted into a one-hot vector. In order to find an optimal classification model by using a C-class dataset, the generic problem is solved by employing the *softmax cross-entropy* function $L_i(w) = -\sum_{i=1}^{C} (y_i)_k \log(h(x_i; w))_k$ for $i = 1, \ldots, N$.

We would like to construct a neural network corresponding to $h(.; w)$ and train it using some data $\{(x_i, y_i)\} \subseteq \{(x_i, y_i)\}_{i=1}^N$ to make a true prediction in image classification. The definition of a neural network is done by specifying an array of layers which creates the specified architecture of a network. This architecture is then established using the MATLAB function `layerGraph` that takes `layers` as an input parameter. Moreover, we would like to make use of training algorithms which are not built-in functions. In this case, we can use a model function `dlnetwork` to define an architecture and customize training loops corresponding user's prescribed algorithm to train it. The `1×1` object `dlnetwork` is a pack of properties including `Layers`, `Connections`, `Learnables`, `State`, `InputNames` and `OutputNames`. We illustrate with an example how to define a `dlnetwork` and show its main properties, i.e., `Layers` and `Learnables`.

Given images with a determined size `inputSize` belonging to a number of classes `numClasses`, we can build a simple network to perform the classification task as follows

```
>> layers = [
>>    imageInputLayer(inputSize,'Normalization','none','Name','input')
>>    convolution2dLayer(5,20, 'Padding', 'same', 'Name', 'conv1')
>>    batchNormalizationLayer('Name', 'bn1')
>>    reluLayer('Name','relu1')
>>    convolution2dLayer(5, 50, 'Padding', 1,'Name', 'conv2')
>>    batchNormalizationLayer('Name', 'bn2')
>>    reluLayer('Name','relu2')
>>    maxPooling2dLayer(2, 'Stride', 2, 'Name', 'maxpool1')
>>    fullyConnectedLayer(numClasses, 'Name', 'fc1')
>>    softmaxLayer('Name', 'softmax')];
>> lgraph = layerGraph(layers);
>> dlNet = dlnetwork(lgraph);
```

This instructions define the different layers that will be applied in a proper sequential order. The function `layerGraph` set the layers as the network's architecture of the `dlnetwork` called `dlNet`. Figure 1 shows the content of different fields of the `dlNet` object. For instance, `dlNet.Layers` contains the network's architecture while `dlNet.Learnables` contains all `Weights` and `Bias` of the convolutional layers, and all `Offset` and `Scale` of the batch normalization layers [2] which constitute the set of trainable parameters (learnables for the `dlNet`). We note that the whole set of values of `dlNet.Learnables` corresponds to parameter vector $w \in \mathbb{R}^n$ in (2.1).

The object `dlNet` should be initialized, that is, an initial value should be given to all weights and bias of each `convolution2dLayer` as well as offset and scale values should be chosen initially for each `batchNormalizationLayer`. The DL toolbox provides some default initializers for `dlNet.Learnables` (and correspondingly for the parameter vector $w \in \mathbb{R}^n$ in (2.1)). In this work, weights and biases are initialized by the Glorot initializer [1] and zeros, respectively, while scale and offsets are, respectively,

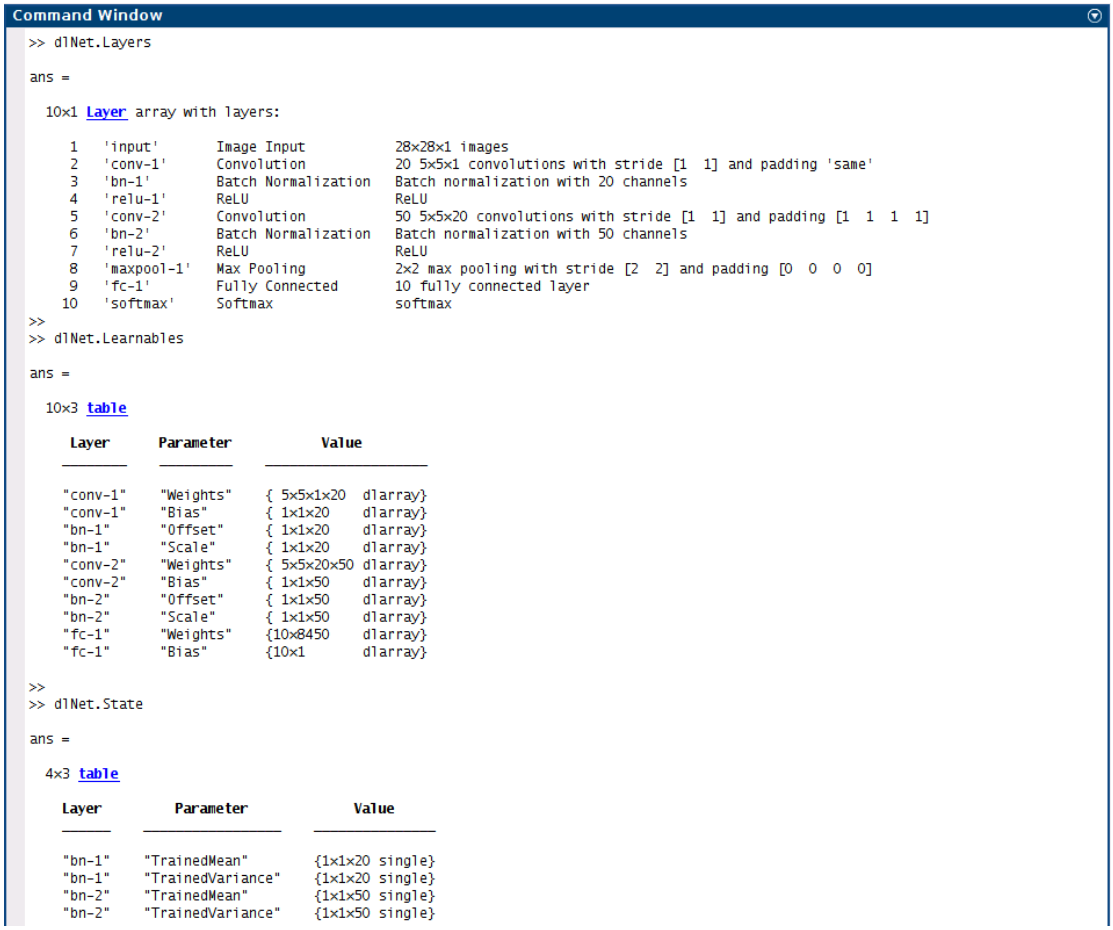

Figure 1. Properties `Layers`, `Learnables` *and* `State` *in* `dlNet`

set to ones and zeros.[1]

Since `dlNet.Learnables` are layered and stored in a specific format (`table`), we should unroll the values of `dlNet.Learnables` to a large parameter vector $w \in \mathbb{R}^n$ in (2.1) to ease computations in the training loop. Moreover, we should use the functions `extractdata` and `gather` for extracting numeric values stored in `dlarray` and collecting data from a GPU (if it is applied), respectively. For instance, the following code can be use to apply this conversion:

```
>> w = [];
>> layeredParam = dlNet.Learnables.Value;
>> for layer = 1: size(layeredParam,1)
>>     val = double(gather(extractdata(layeredParam{layer,1})));
>>     w   = [w; val(:)];
>> end
```

[1]Glorot initializer is also known as Xavier initializer.

2.2. Updating a network

The initial parameter vector w is gradually updated according to an updating rule defined by an optimization algorithm. Such an updating rule, for example, in the quasi-Newton based algorithms such as sL-SR1-TR or sL-BFGS-TR is defined by $w_{k+1} = w_k + p_k$ where w_k is the parameter vector at iteration k and p_k is a search direction obtained by solving $B_k p_k = -g_k$ with Hessian approximation $B_k \in \mathbb{R}^{n\times n}$ and gradient $g_k \in \mathbb{R}^n$. Correspondingly, the *initialized* `dlNet` must be gradually updated using MATLAB `for` or `while` loops to produce a *trained* `dlNet` at the end of the training process; this causes the values of `dlNet.Learnables` to be updated through (`for`- or `while`-) training loops.

Loss function value and gradients are important quantities required per iteration of any gradient-based optimization algorithms such as sL-SR1-TR or sL-BFGS-TR. Given a batch of data denoted by `X` and its true labels `Y`, the initialized network is iteratively trained by iterations composed of a forward propagation pass to compute the overall loss (`loss`) and a backward propagation pass to compute the gradient (`gradient`). To compute `loss` and `gradient`, we use the handle function `dlfeval` in which the functions `forward`, `crossentropy` and `dlgradient` can be called to determine the values of predicted labels, loss and gradient models, respectively. In the DL toolbox, data batch X must be in `dlarray` format and stored in labels `SSCB` that stand for **S**patial, **S**patial, **C**hannel and **B**atch observations. These format and labels enable functions of the DL toolbox to compute derivatives by automatic differentiation. The following piece of code contains the core of the implementation to compute loss and gradient models needed inside the training algorithm:

```
>> dlX = dlarray(single(X),'SSCB');
>> [gradient, loss, state] = dlfeval(@modelgradient, dlNet, dlX, Y);
>>
>> function [gradient, loss, state] = modelgradient(dlNet, dlX, Y)
>>     [dlYp, state]  = forward(dlNet, dlX);
>>     loss           = crossentropy(dlYp, Y);
>>     gradient       = dlgradient(loss, dlNet.Learnables);
>> end
```

The function `dlfeval`, works with `dlNet` and its layered parameters `dlNet.Learnables` in `dlArray` format. As a result, the computed `dlgradient` and `loss` are obtained as a layered gradient variables and a single loss variable both stored in a `dlArray` format, respectively. To obtain the numeric values of both loss and gradient vector we can use, for instance, the following instructions:

```
>> F = double(gather(extractdata(loss)));
>> g = [];
>> layeredGrad = gradient.Value;
>> for layer = 1: size(layeredGrad,1)
>>    val = double(gather(extractdata(layeredGrad{layer,1})));
>>    g   = [g; val(:)];
>> end
```

Given vector g_k denoted by `g` and matrix B_k, solving $B_k p_k = -g_k$ using any proper algorithms for a search direction p_k, denoted by `p` leads to have a numeric vector. Therefore, we should convert the numeric vector `p` into a layered variable in order to be able to update `dlNet.Learnables` for the next iteration, which is equivalent to the

updating rule $w_{k+1} = w_k + p_k$:

```
>> Direction     = dlNet.Learnables;
>> end_array     = 0;
>> for layer = 1: size( Direction, 1)
>>     layer_size     = size( Direction.Value{layer,1} );
>>     start_array    = end_array + 1;
>>     end_array      = end_array + prod(layer_size);
>>     p_segment      = p(start_array : end_array);
>>     tensor         = dlarray(single(reshape(p_segment, layer_size)));
>>     Direction.Value{layer, 1} = tensor;
>> end
>> dlNet.Learnables = dlupdate(@(w,p) w + p, dlNet.Learnables, Direction);
```

2.3. Evaluating a network

To monitor the accuracy of training process, we can use the following statements: [2]

```
>> Yp         = extractdata(dlYp)
>> [~,ind_Yp] = max(Yp, [], 1);
>> [~,ind_Y]  = max(Y, [], 1);
>> accuracy   = mean(ind_Y == ind_Yp);
```

Moreover, given `dlX_v` and `Y_v` as validation examples and their one-hot targets, respectively, the evaluation of the network's performance on the validation dataset can be monitor during the training process as follows:

```
>>  dlYp       = predict(dlNet, dlX_v);
>>  loss       = crossentropy(dlYp, Y_v);
>>  Yp         = extractdata(dlYp)
>>  [~,ind_Yp] = max(Yp, [], 1);
>>  [~,ind_Y]  = max(Yv, [], 1);
>>  acc        = mean(ind_Y == ind_Yp);
```

Batch normalization layers behave differently during the training and inference phases. To set the network to the desired functionality, the functions `forward` and `predict` are used to compute the training and inference outputs, respectively. Specifying `state` as the second output of `forward` during training produces the mean running average $\bar{\mu}$ and the variance running average $\bar{\sigma^2}$ to be computed by (A4) and (A5); see Appendix A. The updated values contained in `TrainedMean` and `TrainedVariance` can be stored in `dlNet.State` in `table` formats (see Figure 1) using:

```
>> dlNet.State = state;
```

At the end of the training step, when the function `predict` employs `dlNet` to make predictions on the validation dataset, the batch normalization layers of the `dlNet` use the final associated `TrainedMean` and `TrainedVariance` stored in `dlNet.State` in place of computing the mean and variance of the validation dataset. Failing to update `state` during training causes the batch normalization layers to use the initial mean and variance in the inference step which results in a poor prediction.

[2] Note that each column of the `Y` denotes a one-hot vector of true label while every column of the `dlYp` is a probability coming from softmax layer. Nevertheless, since the function `max` finds the maximum value and its corresponding location, transforming probabilities into one-hot vectors is not needed.

References

[1] X. Glorot and Y. Bengio, *Understanding the difficulty of training deep feedforward neural networks*, in *Proceedings of the thirteenth international conference on artificial intelligence and statistics*. JMLR Workshop and Conference Proceedings, 2010, pp. 249–256.

[2] S. Ioffe and C. Szegedy, *Batch normalization: Accelerating deep network training by reducing internal covariate shift*, in *International conference on machine learning*. PMLR, 2015, pp. 448–456.

[3] M. Yousefi and A. Martinez Calomardo, *On the efficiency of Stochastic Quasi-Newton Methods for Deep Learning*, preprint, 2022.

Appendix A. Batch normalization

The procedure implemented inside `batchNormalizationLayer` during training includes both standardization and normalization of the output of the previous layer. Let $z_i^{[l]}$ be the ith sample of the current training batch J in the `batchNormalizationLayer` indicated as layer l. The values of this sample vector are standardized as

$$\hat{z}_i^{[l]} = \frac{z_i^{[l]} - \mu}{\sqrt{\sigma^2 + \epsilon}}, \tag{A1}$$

where the scalar ϵ improves numerical stability, and μ and σ^2 are the mean and the variance of the batch J of size bs

$$\mu = \frac{1}{bs}\sum_{i=1}^{bs} z_i^{[l]}, \qquad \sigma^2 = \frac{1}{bs}\sum_{i=1}^{bs}(z_i^{[l]} - \mu)^2, \tag{A2}$$

respectively. After standardization, values $\hat{z}_i^{[l]}$ are normalized by β (`Offset`) and γ (`Scale`) as

$$\tilde{z}_i^{[l]} = \gamma \hat{z}_i^{[l]} + \beta. \tag{A3}$$

The functionality of `batchNormalizationLayer` is different in the inference phase. In this phase, it uses the mean and variance of *trained data* to normalize new data. For this reason, it is necessary to keep track of both quantities during training. Let $X^{\{1\}}$ be the first batch of data used in the initial training iteration of a stochastic algorithm. Let's assume a network with two batch normalization layers, namely, bn1 and bn2, is used, then the mean of $X^{\{1\}}$ is $\mu = [\mu^{\{1\}[1]}, \mu^{\{1\}[2]}]$ where $\mu^{\{1\}[1]}$ is the mean of $X^{\{1\}}$ in bn1 and $\mu^{\{1\}[2]}$ is the mean of $X^{\{1\}}$ in bn2 obtained by (A2). Set $\bar{\mu} = \mu$ as the first *mean running average* for the network trained by $X^{\{1\}}$. Using the second batch $X^{\{2\}}$ in the next training iteration leads to $\mu = [\mu^{\{2\}[1]}, \mu^{\{2\}[2]}]$. Therefore, $\bar{\mu}$ as the *mean running average* for the network trained by $X^{\{1\}}$ and $X^{\{2\}}$ is obtained by

$$\bar{\mu} = \phi\mu + (1 - \phi)\bar{\mu}, \tag{A4}$$

where ϕ denotes the statistic decay value, say 0.1. This process must be continued to the end of the training step. In a similar fashion, the *variance running average* $\bar{\sigma^2}$ is

computed as

$$\bar{\sigma^2} = \phi\sigma^2 + (1 - \phi)\bar{\sigma^2}. \tag{A5}$$

At the end of the training process, the most recent running averages of both statistics, stored in the `TrainedMean` and `TrainedVariance` parameters, are used for normalization in the inference step.