Mini Deep-Learning Framework

Menghe JIN
menghe.jin@epfl.ch

Yueqing SHEN
yueqing.shen@epfl.ch

Jiaan ZHU
jiaan.zhu@epfl.ch

***Abstract*—Though there have been various tools for deep-learning structures, the exploration of this field always starts with manually implementing some simple neural networks. In the project, a simple neural network is constructed with PyTorch tensor operations with components like fully-connected layers, several activation functions, batch normalization and their corresponding back propagation.**

I. Introduction

The goal of this project is to manually implement a simple neural network with some key components, such as fully-connected layer, activation function, batch normalization, etc. In this report, the implementation of these key components and the math behind is shown in Section II and the structure of the Python class is described in Section III. Section IV shows some empirical results and Section V concludes the report.

II. Components

In this report, we denote the row vectors as bold lower-case letters (e.g. $\boldsymbol{b}$), matrices as upper-case letters (e.g. W), multi-dimensional arrays as bold upper-case letters (e.g. $\boldsymbol{W}$), activation functions as $\delta(\cdot)$ and constants as lower-case Greek letters (e.g. γ). For each of the parameters and results, a superscript will be added to show on which layer it is in, like $W^{[l]}$, while another superscript is added to infer the sample index, like $\boldsymbol{x}^{[l],(k)}$. For all subscripts, they are indicating the position of element in the array/matrix/vector (e.g. $W_{ij}^{[l]}$, $\boldsymbol{x}_i^{[l],(k)}$).

A. Fully-connected layer

For fully-connected layers, we use the same dimensions as our code where the input matrix $X^{[l-1]} \in \mathbb{R}^{N\times D^{[l-1]}}$ with each row corresponding to each sample, parameters $W^{[l]} \in \mathbb{R}^{D^{[l-1]}\times D^{[l]}}$ and $\boldsymbol{b}^{[l]} \in \mathbb{R}^{1\times D^{[l]}}$. Then we have

$$Y^{[l]} = X^{[l-1]}W^{[l]} + \boldsymbol{b}^{[l]}, \quad Y^{[l]} \in \mathbb{R}^{N\times D^{[l]}}$$

For each sample represented in a row vector ($\boldsymbol{x}$ or $\boldsymbol{y}$), it is

$$\boldsymbol{y}^{[l],(k)} = \boldsymbol{x}^{[l-1],(k)} \cdot W^{[l]} + \boldsymbol{b}^{[l]}$$

For simpler representation in the derivation of gradient, temporarily rewrite the above formula as

$$\boldsymbol{y} = \boldsymbol{x}W + \boldsymbol{b}$$

and we have

$$\boldsymbol{y}_j = \sum_{i=1}^{D^{[l-1]}} \boldsymbol{x}_i W_{ij} + \boldsymbol{b}_j, \quad j = 1, ..., D^{[l]}$$

To get the gradient in the back propagation, we are looking for $\nabla_{\boldsymbol{x}}\mathcal{L}$, $\nabla_W\mathcal{L}$ and $\nabla_{\boldsymbol{b}}\mathcal{L}$ and by chain rule,

$$\nabla_{\boldsymbol{x}}\mathcal{L} = \left[\sum_{j=1}^{D^l} \frac{\partial\mathcal{L}}{\partial\boldsymbol{y}_j}\frac{\partial\boldsymbol{y}_j}{\partial\boldsymbol{x}_i}\right] = \left[\sum_{j=1}^{D^l} \frac{\partial\mathcal{L}}{\partial\boldsymbol{y}_j}W_{ij}\right] = \nabla_{\boldsymbol{y}}\mathcal{L}\cdot W^T$$

$$\nabla_W\mathcal{L} = \left[\frac{\partial\mathcal{L}}{\partial\boldsymbol{y}_j}\frac{\partial\boldsymbol{y}_j}{\partial W_{ij}}\right] = \left[\frac{\partial\mathcal{L}}{\partial\boldsymbol{y}_j}\boldsymbol{x}_i\right] = \boldsymbol{x}^T\cdot\nabla_{\boldsymbol{y}}\mathcal{L}$$

$$\nabla_{\boldsymbol{b}}\mathcal{L} = \left[\frac{\partial\mathcal{L}}{\partial\boldsymbol{y}_j}\frac{\partial\boldsymbol{y}_j}{\partial\boldsymbol{b}_j}\right] = \left[\frac{\partial\mathcal{L}}{\partial\boldsymbol{y}_j}\right] = \nabla_{\boldsymbol{y}}\mathcal{L}$$

B. 2D convolutional layer

In the implementation of the 2D convolutional layer, we adopt the most straightforward but time-consuming strategy and iterates through all samples, all output channels and all strides. Denoting the input channel number as $C^{[l-1]}$, output channel as $C^{[l]}$ and assume that there's only one sample. Same as the dimensions used in the code, we have

Input:	$\boldsymbol{X} \in \mathbb{R}^{1\times C^{[l-1]}\times D_H\times D_W}$
Weight:	$\boldsymbol{W} \in \mathbb{R}^{C^{[l]}\times C^{[l-1]}\times K_H\times K_W}$
Bias:	$\boldsymbol{b} \in \mathbb{R}^{C^{[l]}}$
Stride:	$\boldsymbol{S} \in \mathbb{R}^{C^{[l-1]}\times K_H\times K_W}$

where strides are windows from the input that involves the element-wise product with the weights. Denoting the padding as p and stride step as s, the output would be of shape

$$\left(1, C^{[l]}, 1+\frac{(D_H+2p-K_H)}{s}, 1+\frac{(D_W+2p-K_W)}{s}\right)$$

and the output box at position $(1, c, h, w)$ is computed as

$$\boldsymbol{Y}_{(1,c,h,w)} = \sum \boldsymbol{S}_{(1,h,w)} \otimes \boldsymbol{W}_c + \boldsymbol{b}_c$$

where c is the index for channel, h and w are indices for strides and $\otimes$ stands for element-wise multiplication.

It can be observed that each stride and parameter is related to more than one values in the output array, which indicates that their gradients in the back propagation should be sum over channels or strides if the algorithm iterates over all elements in the output array. Therefore, our implementation goes like

$$\nabla_{\boldsymbol{S}_{(1,h,w)}}\mathcal{L} = \sum_c \boldsymbol{W}_c \otimes \nabla_{\boldsymbol{Y}_{(1,c,h,w)}}\mathcal{L}$$

$$\nabla_{\boldsymbol{W}_{(c)}}\mathcal{L} = \sum_h\sum_w \boldsymbol{S}_{(1,h,w)} \otimes \nabla_{\boldsymbol{Y}_{(1,c,h,w)}}\mathcal{L}$$

$$\nabla_{\boldsymbol{b}_{(c)}}\mathcal{L} = \sum_h\sum_w \nabla_{\boldsymbol{Y}_{(1,c,h,w)}}\mathcal{L}$$

C. Batch normalization

To perform batch normalization, again temporarily ignore the layer superscripts, in the forward pass of training, we have

$$\begin{aligned}
\boldsymbol{\mu}_j &= \frac{1}{N}\sum_{k=1}^{N} Y_{kj}, \quad j = 1, ..., D^{[l]} \\
\boldsymbol{\sigma}_j^2 &= \frac{1}{N}\sum_{k=1}^{N} \left(Y_{kj} - \boldsymbol{\mu}_j\right)^2, \quad j = 1, ..., D^{[l]} \\
\bar{Y}_{kj} &= \frac{Y_{kj} - \boldsymbol{\mu}_j}{\sqrt{\boldsymbol{\sigma}_j^2 + \epsilon}}, \quad k = 1, ..., N, \quad j = 1, ..., D^{[l]} \\
\hat{Y}_{kj} &= \boldsymbol{\gamma}_j \cdot \bar{Y}_{kj} + \boldsymbol{\beta}_j, \quad j = 1, ..., D^{[l]}
\end{aligned}$$

where N is the number of samples; $\boldsymbol{\mu}_j$ and $\boldsymbol{\sigma}_j^2$ are the mean and variance of characteristics at index j; $\boldsymbol{\gamma}$ and $\boldsymbol{\beta}$ are to be learned. In the forward pass of testing, however, the normalized data is not calculated from the sample mean and variance, but the running mean and variance which has been updated at each iteration of training, with

$$\begin{aligned}
\hat{\boldsymbol{\mu}}_j &= \rho\hat{\boldsymbol{\mu}}_j + (1-\rho)\boldsymbol{\mu}_j \\
\hat{\boldsymbol{\sigma}}_j &= \rho\hat{\boldsymbol{\sigma}}_j + (1-\rho)\boldsymbol{\sigma}_j
\end{aligned}$$

In back propagation of batch normalization,

$$\begin{aligned}
\nabla_{\bar{Y}}\mathcal{L} &= \left[\frac{\partial\mathcal{L}}{\partial\bar{Y}_{kj}}\right] = \left[\frac{\partial\mathcal{L}}{\partial\hat{Y}_{kj}}\frac{\partial\hat{Y}_{kj}}{\partial\bar{Y}_{kj}}\right] = \left[\frac{\partial\mathcal{L}}{\partial\hat{Y}_{kj}}\boldsymbol{\gamma}_j\right] \\
&= (\boldsymbol{\gamma})^T \otimes \nabla_{\hat{Y}}\mathcal{L} \quad \text{(with broadcasting)}
\end{aligned}$$

$$\begin{aligned}
\nabla_{\boldsymbol{\gamma}}\mathcal{L} &= \left[\sum_{k=1}^{N}\frac{\partial\mathcal{L}}{\partial\hat{Y}_{kj}}\frac{\partial\hat{Y}_{kj}}{\partial\boldsymbol{\gamma}_j}\right] = \left[\sum_{k=1}^{N}\frac{\partial\mathcal{L}}{\partial\hat{Y}_{kj}}\bar{Y}_{kj}\right] \\
&= \sum_{k=1}^{N}\left(\nabla_{\hat{Y}}\mathcal{L}\otimes\bar{Y}\right)_k
\end{aligned}$$

$$\begin{aligned}
\nabla_{\boldsymbol{\beta}}\mathcal{L} &= \left[\sum_{k=1}^{N}\frac{\partial\mathcal{L}}{\partial\hat{Y}_{kj}}\frac{\partial\hat{Y}_{kj}}{\partial\boldsymbol{\beta}_j}\right] = \left[\sum_{k=1}^{N}\frac{\partial\mathcal{L}}{\partial\hat{Y}_{kj}}\right] \\
&= \sum_{k=1}^{N}\left(\nabla_{\hat{Y}}\mathcal{L}\right)_k
\end{aligned}$$

where $\otimes$ stands for element-wise multiplication. And for the gradient with respect to Y,

$$\begin{aligned}
\nabla_{\boldsymbol{\sigma}^2}\mathcal{L} &= \left[\partial\mathcal{L}/\partial\boldsymbol{\sigma}_j^2\right] \\
&= \left[-\frac{1}{2}\sum_{k=1}^{N}\frac{\partial\mathcal{L}}{\partial\bar{Y}_{kj}}\left(Y_{kj}-\boldsymbol{\mu}_j\right)\left(\boldsymbol{\sigma}_j^2+\epsilon\right)^{-3/2}\right] \\
&= -\frac{1}{2}\sum_{k=1}^{N}\left(\nabla_{\bar{Y}}\mathcal{L}\otimes(Y-\boldsymbol{\mu})\right)_k \otimes \left(\boldsymbol{\sigma}^2+\epsilon\right)^{-3/2}
\end{aligned}$$

$$\begin{aligned}
\nabla_{\boldsymbol{\mu}}\mathcal{L} &= \left[\partial\mathcal{L}/\partial\boldsymbol{\mu}_j\right] = \left[-\sum_{k=1}^{N}\frac{\partial\mathcal{L}}{\partial\bar{Y}_{kj}}\left(\boldsymbol{\sigma}_j^2+\varepsilon\right)^{-1/2}\right] \\
&\quad + \left[\frac{\partial\mathcal{L}}{\partial\boldsymbol{\sigma}_j^2}\cdot\frac{1}{N}\cdot\sum_{k=1}^{N}(-2)\cdot\left(Y_{kj}-\boldsymbol{\mu}_j\right)\right] \\
&= -\sum_{k=1}^{N}\left(\nabla_{\bar{Y}}\mathcal{L}\right)_k \otimes \left(\boldsymbol{\sigma}^2+\varepsilon\right)^{-1/2} \\
&\quad - 2\cdot\nabla_{\boldsymbol{\sigma}^2}\mathcal{L}\otimes\left(\frac{1}{N}(Y-\boldsymbol{\mu})\right)
\end{aligned}$$

$$\begin{aligned}
\nabla_{Y}\mathcal{L} &= [\partial\mathcal{L}/\partial Y_{kj}] = \left[\frac{\partial\mathcal{L}}{\partial\bar{Y}_{kj}}\cdot\left(\boldsymbol{\sigma}_j^2+\epsilon\right)^{-1/2}\right] \\
&\quad + \left[\frac{\partial\mathcal{L}}{\partial\boldsymbol{\sigma}_j^2}\frac{2\left(Y_{kj}-\boldsymbol{\mu}_j\right)}{N} + \frac{\partial\mathcal{L}}{\partial\boldsymbol{\mu}_j}\cdot\frac{1}{N}\right] \\
&= \nabla_{\bar{Y}}\mathcal{L}\otimes\left(\boldsymbol{\sigma}^2+\epsilon\right)^{-1/2} \\
&\quad + \frac{2}{N}\nabla_{\boldsymbol{\sigma}}\mathcal{L}\otimes(Y-\boldsymbol{\mu}) + \frac{1}{N}\nabla_{\boldsymbol{\mu}}\mathcal{L}
\end{aligned}$$

D. Activation functions

We have implemented ReLU, sigmoid and tanh as activation functions ($Z = \delta(\hat{Y})$) and their gradients are implemented as: For ReLU,

$$\nabla_{\hat{Y}_{kj}}\mathcal{L} = \begin{cases} \nabla_{Z_{kj}}\mathcal{L}, & \text{for } Y_{kj} \geq 0 \\ 0, & \text{for } Y_{kj} < 0 \end{cases}$$

For tanh,

$$\nabla_{\hat{Y}_{kj}}\mathcal{L} = \nabla_{Z_{kj}}\mathcal{L}\cdot\left(1 - Z_{kj}^2\right)$$

and for sigmoid,

$$\nabla_{\hat{Y}_{kj}}\mathcal{L} = \nabla_{Z_{kj}}\mathcal{L}\cdot Z_{kj}\cdot(1 - Z_{kj})$$

III. Structure of model

We have implemented two Python classes, one being the inheritance of the other. The parent class, `Module`, can be configured as a single Module or a sequence of Modules. A single Module includes a fully-connected/2D-convolutional layer, an optional batch normalization layer and an optional activation function, which should all be specified in the constructor of a Module instance together with the shape of the weights. The 'Sequential' type Module contains a list of single Modules in the order of appending them by a public method named `append(Module)`, as illustrated in Fig.1. Module class has several other public methods, `forward`,`loss`, `backward` and `updata_params`. These methods takes a mini-batch of the whole dataset, computing the loss and its gradient, performing back propagation and gradient descent on the parameters.

Module class has several other public methods, `forward`, `loss`, `backward` and `updata_params`. These methods takes a mini-batch of the whole dataset, computing the mean squared error as the loss and its gradient, performing back propagation and gradient descent on the parameters. The child class, `NeuralNet`, takes extra parameters for the training

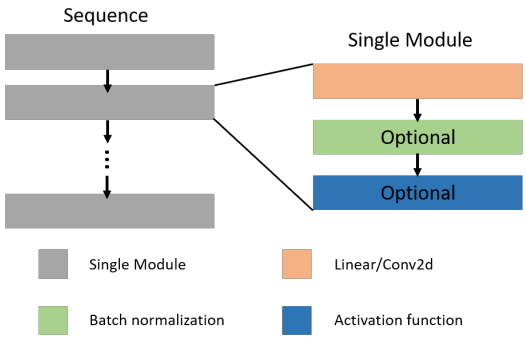

Fig. 1: Illustration of `Module` class

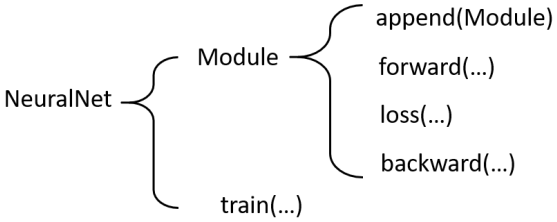

Fig. 2: Structure of `NeuralNet` class

process, such as the learning rate, the regularization term, the number of epoches and the mini-batch size. The stochastic gradient descent(SGD) is implemented in the method `train` and is performed by randomly picking a mini-batch of samples for forward and backward pass instead of a single sample. The structure of the two classes is illustrated in Fig.2.

IV. Experiments and Results

The training and testing dataset is generated with 1000 samples uniformly distributed in $[0,1]^2$, each with alabel 0 if outside the disk centered at $(0.5, 0.5)$ of radius $1/\sqrt{2\pi}$ and 1 inside. A sanity check is make by scattering the data points and the disk, as shown in Fig.3.

The neural network in the trial is a sequence of 4 linear Modules with ReLU activation, first 3 of which adopts batch normalization. The shape of weights in each layer is $(2, 100)$, $(100, 50)$, $(50, 20)$ and $(20, 1)$ respectively. Training has been performed over 1000 epoches, with learning rate of $2e^{-3}$, regularization of 0.05 and mini-batch size of 64. Fig.4 is an example of the loss change over the training and the accuracy on the testing data for this set of losses is around 0.977. Most training proceeds as above but there are occasionally some less satisfying training results with slower convergence and an accuracy of 0.74.

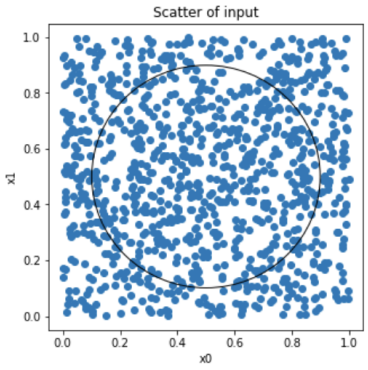

Fig. 3: Validation of the generated dataset

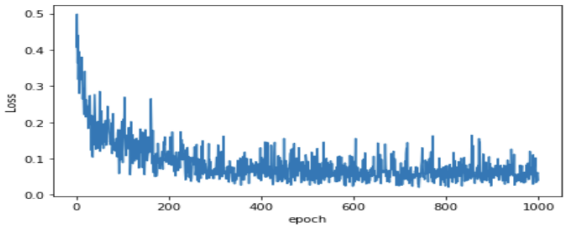

Fig. 4: Losses over training

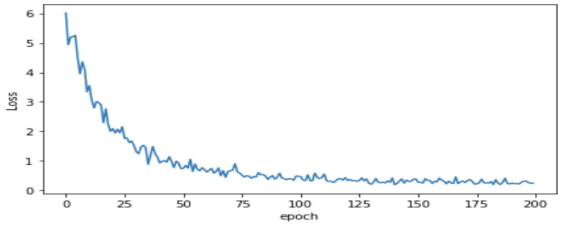

Fig. 5: Losses over training with convolutional layers

Additionally, we constructed a network with 2 convolutional layers and 2 fully-connected layers. Each input samples is in $[0,1]^{3\times4\times4}$ and the criteria for 1 is that the 2-norm of $\boldsymbol{x} - 0.5^{3\times4\times4}$ should be smaller than 2. The shapes of the weights are set to be $(5,3,3,3)$, $(1,5,3,3)$, $(16,10)$ and $(10,1)$. The training loss with learning rate of $3e^{-3}$, regularization of 0.05, mini-batch size of 64 is shown in Fig.5 and the accuracy is around 0.7. It is noticed that the CNN performs well on GPUs while generates NaN values on CPUs. Existing tools such as Keras may have such problem as well. This is not solved in our project, so the file *test.py* only contains fully-connected layers and the above results is produced on an RTX2060 GPU.

V. Conclusion

In this project we have implemented some basic components of a neural network. The model is capable of training on the data with expected performances. This project may become a starting point of construction other components and structures, such as max-pooling and dropout, and some optimization of algorithm.