Deep Learning Project 2

Jacob Bamberger (328027)**, Kristians Cernevics** (293870)**, Mariella Kast** (271602)

November 29, 2021

1 Introduction

Machine learning libraries are usually an indispensable compilation of routinely used functions and modules. Hence, libraries like PyTorch [3], TensorFLow [1], JAX [2] and many more allows users to write complex programs and save time by avoiding re-writing redundant code many times. Here, we create our own mini deep learning library utilizing few of the most common modules such as linear layer, multiple activation functions, mean squared loss and sequential. The report is structured as follows: first we discuss the design of our modules, then training implementation and finally, we discuss the performance on the classification task.

2 Design of Modules

2.1 Module

As suggested in the project description, we started with implementing an object that all other Modules inherit from. The methods include *forward*, *backward*, *param*, and *SGD_step* methods. The first two are essential and need to be implemented in each module, while the other two return default values (empty list or None) in case they are not applicable for the module.

2.2 Linear

The Linear (or fully connected) layer module is used to apply an affine transformation to the input. The module is initialized by providing the dimensions of both the input and the output and, optionally, a string corresponding to a weight initialization method. Upon initialization, two tensors are stored corresponding to the weights and the biases of the model. In addition, the accumulated gradient tensor is created for both parameters. This module thus saves the parameters, gradients as well as input and output dimensions for further use.

The *forward* function saves the input tensor and then applies the linear transformation to return it as an output. Our function can deal with both single and mini-batches of inputs. Next, *backward* function requires gradient of loss w.r.t to the *forward* output as an input. This function computes a new gradient of loss w.r.t. to both parameters that is then accumulated in the previously created gradient tensors. Finally, this function returns the gradient of loss w.r.t. to the *forward* input. The function *param* can be used to access the weight, accumulated gradient of weight, bias and accumulated gradient of bias numerical values. For the description of *SGD_step* function, please refer to subsection 2.5.

One of the main design decisions was the weight initialization. *Uniform* or *Normal* distribution are available for parameter (weight w and bias b) initialization.

2.3 Sequential

The sequential module allows combining multiple linear layers and activation functions in order to facilitate deeper network implementation. This module is initialized by providing a tuple of layers to build the network. The *forward* function requires a torch tensor as an input and then calls each layer's *forward* function in a sequence to compute the new output. Similarly, *backward* requires the gradient of loss w.r.t to output as an input and calls each layer's *backward* function in a reverse order to compute the new gradient of loss. Finally, function *SGD_step* requires a float as an input for the learning rate, which is then passed to each layer's *SGD_step* function to update the parameters. Note that except for the sub layers, nothing is stored in this object. Function *param* calls each layer's *param* function and adds the output to a list, which is then returned.

2.4 Tanh, ReLu and Sigmoid

The Tanh module implementation is straightforward as only *forward* and *backward* functions need to be defined. The *forward* function takes one argument as an input, saves it as a variable for further use and then computes the tanh values by utilizing the $Tensor.tanh()$ method for the output. *Backward* function also requires only one input, which should be the gradient of the loss w.r.t the output computed by the next layer or MSE module. To compute the output (gradient of loss w.r.t. to the activation) of the *backward* function we do a component-wise multiplication of the derivative of tanh and the input. As there are no parameters, the *param* and *SGD_step* are not altered. The ReLu and Sigmoid modules are implemented similarly, although it was somewhat easier to store the output of these layers to save computation time in the backward pass.

2.5 MSE

Similar to the activation functions, the MSE module only has *forward* and *backward* functions, where both of them require two input arguments - the prediction and the target. While the *forward* function simply returns the mean squared error, the *backward* returns the derivative. MSE module can deal with both individual and mini-batches of inputs.

3 Training implementation

3.1 SGD

The stochastic gradient descent is implemented as a function on Modules. Since the only Module that has parameters is *Linear*, the *SGD_step* function simply returns None for all other modules. We decided against implementing the SGD as a separate module as we are only dealing with the Linear layer as the sole module that has optimizable parameters. The *SGD_step* function requires one input as the learning rate η. To update the weights and bias parameters, we multiply the accumulated gradient w.r.t to weights and bias with the learning rate η and add it to the existing parameters. After this step, we set the accumulated gradients to zero.

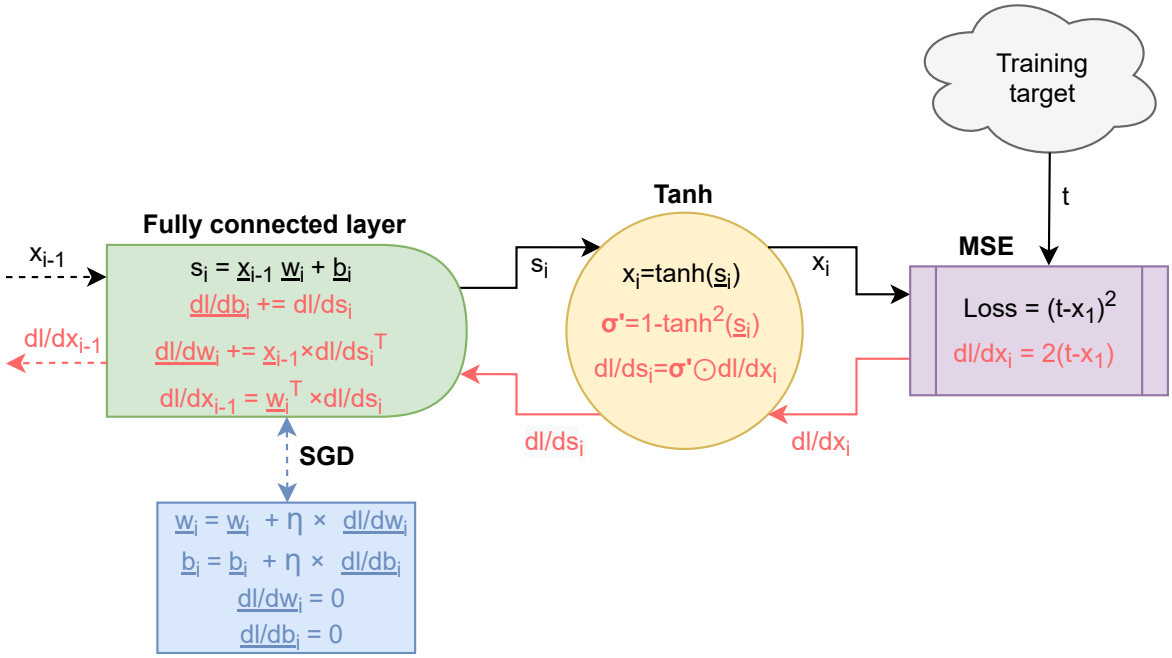

Figure 1: Scheme of fully connected layer with a Tanh activation, mean squared error loss computation and stochastic gradient descent optimizer. Black arrows and equations indicate the forward pass, while red arrows and equations indicate backward pass. Stochastic gradient descent implementation is indicated by the color blue. Variables with an underscore are saved in the corresponding module.

3.2 Forward pass

We present a sample forward pass for a fully connected layer with Tanh activation in Figure 1 marked with black arrows and black equations. Mean squared error (MSE) module is used to compute the final loss. Additional layers may be added to the left side of the scheme without losing generality. Other activation functions such as ReLu or Sigmoid can be easily used to replace the Tanh module, with the only changes being in the computation x_i from s_i and the saved variables (output instead of the input for Tanh module).

3.3 Backward pass

Similarly, we show a scheme of a backward pass for a fully connected layer with Tanh activation in Figure 1 marked with red arrows and red equations, with the direction going from right to left side of the figure. The first step of the backward pass starts with the calculation of the gradient of the loss dl/dx_i in the MSE module, which depends on the forward pass output x_i and training target t. Next, the dl/dx_i is used as an input for the Tanh module, where we first compute the gradient of the Tanh function and finally the gradient of loss w.r.t. the summation before activation dl/ds_i. Furthermore, we utilize dl/ds_i as an input for the fully connected layer, where we have to compute the gradient of loss w.r.t. parameters w_i, b_i and, finally, the gradient of loss w.r.t. the input x_i. One can notice that the final output of the backwards pass is dl/dx_{i-1}, hence showing the generality of our calculation and the extendibility of our network. In general, we see that the backward pass is a simple computation that depends on applying the chain rule again and again. However, the backward pass requires us to keep trace of both inputs x_i and pre-non-linearity activations s_i, hence these quantities to be saved memory, and is computationally twice more expensive than the forward pass as we have to calculate both dl/dx_{i-1} and dl/dw_i.

3.4 Batches

Our first implementation followed the practical 3 closely and operated on a single data sample at a time, leading to long computation times. Hence, we now process the samples in mini-batches, utilizing matrix multiplications instead of a for loop. This requires small, but subtle changes in the code in order to deal with the extra dimension in the input. For example, we could no longer use matrix-vector multiplication and hence adaptation of the 1D input was required. This change ultimately accelerated the calculations, e.g. the usage of mini-batches sizes of 50 saw a speed-up by a factor of 60.

4 Performance on the classification task

In this section we discuss performance of our framework on the classification task consisting of classifying whether random points in the unit square of the plane belong to the unit circle in the plane. The target is labelled with 0 if the point is outside the disk, and 1 if inside. In order for our predictions to be between 0 and 1, we decided to make our last layer a *Sigmoid* activation. We thus considered several different networks, all with three hidden layers of 25 units each, but varying in intermediate activation functions and weight initialization. After trying several learning rates, we found that a learning rate of 0.01 was quasi-optimal for each network. We trained each network on 1000 samples and tested them on 1000 samples as well. In Table 1, we report the average and standard deviation statistics for time, final train error and test error of each networks on 15 separate runs, each trained for 250 epochs.

Our results show no significant difference between most networks, with a few exceptions. Both ReLu networks are significantly faster than the other networks, which is likely due to the simplicity of the ReLu layer which does not require multiplication of two small numerical values for the backward pass. Ironically, the Sigmoid network is both he worst and the best one, depending on weight initialization.

	ReLu (normal)	ReLu (uniform)	Tanh (normal)	Tanh (uniform)	Sigmoid (normal)	Sigmoid (uniform)
Time (s)	1.82 ± 0.01	1.87 ± 0.09	2.50 ± 0.23	2.12 ± 0.19	2.07 ± 0.16	2.13 ± 0.12
Train error (%)	0.84 ± 0.41	0.85 ± 0.22	0.77 ± 0.17	1.7 ± 0.90	0.48 ± 0.21	49.9 ± 0.00
Test error (%)	1.95 ± 0.28	1.87 ± 0.22	1.77 ± 0.27	2.07 ± 0.81	1.09 ± 0.23	50.1 ± 0.00

Table 1: Statistics of the different models for the classification task, all trained for 250 epochs, batch size 50, with MSE loss, and learning rate 0.01

References

[1] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[2] J. Bradbury, R. Frostig, P. Hawkins, M. J. Johnson, C. Leary, D. Maclaurin, G. Necula, A. Paszke, J. VanderPlas, S. Wanderman-Milne, and Q. Zhang. JAX: composable transformations of Python+NumPy programs, 2018.

[3] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.