Creation of a deep learning framework using PyTorch Tensors

Cleres David
Computational Science & Engineering
SCIPER 247182
Email: david.cleres@epfl.ch

Lesimple Nicolas
Computational Science & Engineering
SCIPER 237699
Email: nicolas.lesimple@epfl.ch

Rensonnet Gaëtan
Signal Processing Lab (LTS5)
SCIPER 276950
Email: gaetan.rensonnet@epfl.ch

***Abstract*—In this EE-559 Deep Learning class project, we demonstrate our ability to design an operational prototype of a deep learning in Python. We provide user-friendly routines for the easy instantiation, training and testing of deep architectures resembling multi-layer perceptrons (MLP). Drawing inspiration from the open-source PyTorch project, fully-connected layers and non-linear activation layers are easily assembled using the `Module.Sequential` container. The mean squared error (MSE), negative log-likelihood (NLL) and cross-entropy (CE) loss functions are implemented and stochastic gradient descent (SGD) is provided in its vanilla form as well as with a momentum term. Backward and forward methods handle batch data, leveraging the power of vectorized computations with PyTorch `Tensors`. Similarly to PyTorch, the bulk of the training is done by the `Module.backward()`, `Loss.backward()` and `Optimizer.step()` methods. We validate our framework on a toy-example of binary classification for points inside or outside a disk in a 2D plane. Although the results are very sensitive to meta-parameters, our results are globally shown to be comparable to those of PyTorch, both in accuracy and in efficiency.**

I. Introduction

As self-driving cars clock up millions of miles [1], [3], [4] and Google DeepMind's AlphaGo Zero network beats world champions at the game of Go [6], there is little doubt that artificial intelligence and in particular deep learning will play an increasingly important role in a range of engineering fields. It is therefore important to get acquainted with the nuts and bolts of deep architectures and to understand the phenomena at play inside the 'black box'.

The goal of this project was to design a small deep learning framework from scratch using only PyTorch `Tensor` operations and the standard `math` library from Python. Section II first describes the architecture of the framework and details each of its four main blocks. It then covers the implementation details of the more mathematically-involved operations. It finally describes the four experiments performed to validate the accuracy of the framework and examine its efficiency. Section III presents the results and a final discussion is provided in Section IV.

II. Methods

A. Framework architecture

As depicted in Fig. 1 our framework is composed of three main base classes, `Module`, `Loss` and `Optimizer` and one smaller class named `Parameter`, which are described in the following paragraphs. For any `Module` layer or `Loss` loss function, batch inputs are noted as $X \in \mathbb{R}^{B\times N_{in}}$ and batch outputs as $Y \in \mathbb{R}^{B\times N_{out}}$, where B is the batch size and N_{in}, N_{out} the dimensions of the input and output data, respectively. Individual batch samples are considered as column vectors and denoted by $x^n \in \mathbb{R}^{N_{in}}$ and $y^n \in \mathbb{R}^{N_{out}}$, with $1 \leq n \leq B$.

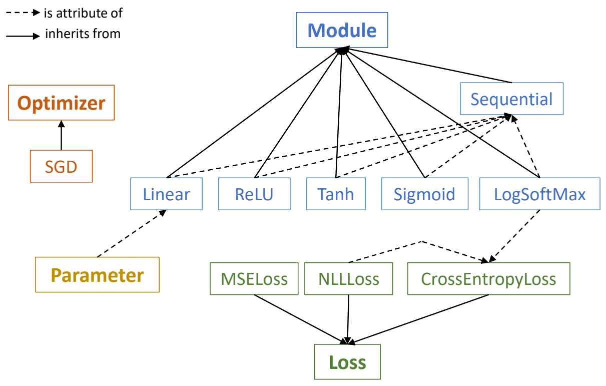


Fig. 1: Main modules of our framework and their inter-dependencies. Bases classes are in bold font.

1) Module layers: Each elementary layer of a network inherits from the `Module` base class and is responsible for performing the forward and the backward pass as well as keeping track of its parameters, if any. This is illustrated in the top row of Fig. 2 and in the code snippet in Fig. 3. In order to perform the backward pass, a `Module` object must store the last input X the forward pass was called on.

The fully-connected linear layer, `Linear`, is the only layer with parameters to be estimated, namely a matrix $W \in \mathbb{R}^{N_{out}\times N_{in}}$ and an (optional) bias $b \in \mathbb{R}^{N_{out}}$. In such a layer, every component of an output y^n is affected by all the components of the corresponding input x^n, which is also true of the `LogSoftMax` layer. Care must therefore be taken for backward pass, as detailed below in Section II-B. There is however no interaction between different batch samples. The non-linear activation layers `ReLU`, `Tanh` and `Sigmoid` perform component-wise operations and are simpler to handle.

2) Loss functions: Loss functions inherit from the `Loss` base class. As illustrated in the code snippet in Fig. 4, their `Loss.loss` method computes the total loss associated with the output Y of a neural network and the associated reference target T. The `Loss.backward` method computes the derivative of the batch loss with respect to each entry of Y and returns a `Tensor` of the same size as Y. Unlike `Module` layers, they do not need to store the latest input values.

The MSE loss is straightforward to implement as it is differentiable and treats all the components of a batch sample identically. The NLL and CE losses proceed slightly differently and will be addressed in Section II-B.

3) Optimizers: As illustrated in Fig. 5, the main responsibility of an `Optimizer` object is to perform an update of all the parameters of the model, also known as a 'gradient step' (although it may involve more than just the gradient accumulated on the current batch), via the `Optimizer.step` method. To that end, an `Optimizer` stores a list of references to all the parameters of the model to be optimized (see `Parameter` class below).

4) Parameters: The `Parameter` class is a simple container for two PyTorch `Tensor` objects: `data`, which contains the parameter itself, and `grad`, which is of the same size as `data` and contains the accumulated gradient of the loss with respect to the parameter. The gradient is reset to zero by the `Parameter.zero_grad` method, which is done before applying the backward pass on a new batch during training.

B. Implementation details

1) Fully-connected linear layer: Mathematically, the forward pass in batch form is expressed as

$$Y = XW^T + \mathbb{1}_B b^T$$

where $\mathbb{1}_B$ is the column vector of all ones in $\mathbb{R}^B$. With PyTorch `Tensors` however, broadcasting allows $Y = XW^T + b^T$ to work similarly. In the backward pass, the derivative of the batch loss l with respect to W_{ij} is

$$\frac{\partial l}{\partial W_{ij}} = \sum_{n=1}^{B} \frac{\partial l}{\partial y_i^n} \frac{\partial y_i^n}{\partial W_{ij}} = \sum_{n=1}^{B} \frac{\partial l}{\partial y_i^n} x_j^n .$$

This can be performed efficiently by reshaping the $\partial l/\partial Y$ `Tensor` passed to `Module.backward` from $B \times N_{out}$ to $B \times N_{out} \times 1$ and the X `Tensor` (stored during the forward pass) from $B \times N_{in}$ to $B \times 1 \times N_{in}$. Element-wise multiplication with implicit broadcasting is then performed followed by summing the B elements along the first dimension, which allows the parameter gradients to be updated. To compute $\partial l/\partial X$ (of size $B \times N_{in}$), we note that

$$\frac{\partial l}{\partial x_j^n} = \sum_{i=1}^{N_{out}} \frac{\partial l}{\partial y_i^n} \frac{\partial y_i^n}{\partial x_j^n} = \sum_{i=1}^{N_{out}} \frac{\partial l}{\partial y_i^n} W_{ij},$$

which in batch form can be performed as the simple matrix multiplication $\partial l/\partial X = (\partial l/\partial Y) \cdot W$. Finally, we chose to initialize the weights and biases by sampling from a uniform distribution with standard deviation $1/N_{in}$ in order to control the variance of the activations (i.e. the inputs and outputs of the layer). Unlike in PyTorch, we included the $\sqrt{3}$ factor stemming from the fact that $\sigma(X) = a/\sqrt{3}$ if $X \sim \mathcal{U}[-a, a]$. The user also has the possibility to specify the non-linear activation function ('relu', 'tanh', 'sigmoid') used in conjunction with the `Linear` layers as these affect the variance of the activations by different factors [2].

2) Log-softmax layer: In this `Module` layer, $N_{in} = N_{out} = N$ and for each batch sample x^n, the output y^n is defined as

$$y_i^n = \log\left(\exp\left(x_i^n\right)/S^n\right) = x_i^n - \log\left(S^n\right), \quad 1 \le i \le N,$$

where $S^n = \sum_{k=1}^{N} \exp\left(x_k^n\right)$. Based on

$$\frac{\partial y_i^n}{\partial x_j^n} = \delta_{ij} - \exp\left(x_j^n\right)/S^n,$$

where $\delta_{ij} = 1$ if $i = j$ and 0 otherwise, we obtain

$$\frac{\partial l}{\partial x_j^n} = \sum_{i=1}^{N} \frac{\partial l}{\partial y_i^n} \frac{\partial y_i^n}{\partial x_j^n} = \frac{\partial l}{\partial y_j^n} - \frac{\exp\left(x_j^n\right)}{S^n} \left(\sum_{i=1}^{N} \frac{\partial l}{\partial y_i^n}\right), \quad (1)$$

from which we can compute $\partial l/\partial X$ using efficient `Tensor` operations.For numerical stability, we compute the term $\exp\left(x_j^n\right)/S^n$ in Eq. (1) by first shifting every x_i^n by $\max_k x_k^n$. This does not affect the result and avoids evaluating exponentials at large values by shifting their arguments to negative values.

3) NLL loss: The NLL batch loss l for the batch output Y of a neural network is defined as

$$l = -\frac{1}{B} \sum_{n=1}^{B} y_{l(n)}^n,$$

where $l(n)$ is the label of the training sample x^n in the input batch X. The derivative $\partial l/\partial Y$ is therefore computed as $\partial l/\partial y_i^n = -\delta_{il(n)}/B$.

4) CE loss: The cross-entropy loss consists in combining a final log-softmax layer for the network with an NLL loss function. Consequently, a `CrossEntropyLoss` object stores a `LogSoftMax` and an `NLLLoss` object as attributes. The forward and backward passes of each of the attributes are then performed successively at each pass. This is done in a transparent manner as storing the input X during each forward pass is done automatically by the `LogSoftMax` attribute for instance.

5) SGD with momentum: We implemented the SGD optimizer both in vanilla form and with a momentum or inertia term as described in [5]. In the latter case, the gradient step performed in the previous iteration must be stored in an internal buffer by the `SGD` object, which was done using a dictionary hashed by the individual `Parameters` of the stored `Parameter` list.

C. Validation on a toy example

We considered a training and a test set of $N_{data} = 1000$ points each sampled uniformly in $[0,1] \times [0,1]$. A point was assigned a label 0 if it fell outside the disk of radius $1/\sqrt{2\pi}$ and a label 1 elsewhere, which ensured a balanced class representation. We constructed a neural network with 2 input units (x and y coordinates), 2 output units (label 0 and label 1) and three 'hidden' fully-connected linear layers containing 25 units each.

The only preprocessing we performed was standardizing the input data: the component-wise mean of the train data was subtracted from the train and the test data, then the train and test samples were divided by the component-wise standard deviation of the train data. This ensured that the range of the data was centered around zero with variance close to 1, which in turn allowed the activations of the network to maintain a variance close to 1, provided a proper parameter initialization was performed (see Section II-B1). One-hot encoding was performed on the label vectors whenever MSE loss was used. Care was taken throughout our code to reduce the hassle of switching from one-hot encoding and back.

In Experiments I-IV described below, the train and test error rates were examined in a variety of scenarios and systematically compared to a similar network built from the `nn` library in PyTorch, taken as a reference. Experiments V and VI focused on the computational efficiency of the framework. We noted η the learning rate, β the momentum gain, E the number of epochs used in the training and B the mini-batch size.

Experiment I. Effect of momentum on classification error: The momentum gain β was varied from 0 to 0.9 with fixed $\eta = 5 \times 10^{-3}$, $E = 100$ and $B = 100$, using the ReLU activation between linear layers. The mean and standard deviation of the train and test error were computed over 15 independent runs, for both our network and its PyTorch counterpart and using both the MSE and the CE loss.

Experiment II. Effect of learning rate on classification error: The learning rate η was varied logarithmically from 1×10^{-5} to 1 with fixed $\beta = 0$, $E = 100$ and $B = 100$, using the MSE loss. The mean and standard deviation of the train and test error were computed over 15 independent runs, for both our network and its PyTorch counterpart and using both the ReLU and the Tanh activation between linear layers.

Experiment III. Effect of number of epochs on classification error: The number of epochs E was varied from 20 to 250 with fixed $\eta = 5 \times 10^{-3}$, $\beta = 0$ and $B = 100$, using the ReLU activation between linear layers. The mean and standard deviation of the train and test error were computed over 15 independent runs, for both our network and its PyTorch counterpart and using both the MSE and the CE loss.

Experiment IV. Effect of parameter initialization on classification error: Instead of the parameter initialization described in Section II-B1, we initialized all the parameters of the linear layers by drawing values from a uniform distribution $\mathcal{U}[-a, a]$ with a taking values in []. We forced the same initialization with the PyTorch network, fixed $\eta =$, $\beta = 0$, $E =$ and $B =$, used ReLU activations and MSE loss and repeated each experiment 15 times.

Experiment V. Effect of batch size on computation time: The batch size was varied from 1 to $N_{data} = 1000$ with fixed $\eta = 1 \times 10^{-2}$, $\beta = 0$ and $E = 50$, using the ReLU activation between linear layers and the MSE loss. The mean computation time was computed over 15 independent runs. The speedup compared to the case $B = 1$ was also reported as a function of B.

Experiment VI. Effect of vectorizing tensor operations on computation time: The computation time of our framework was compared to a naive implementation using Python for-loops for all tensor operations and processing one individual input at a time. To ensure a fair comparison, the batch size was fixed to $B = 1$ in the optimized framework. Other fixed parameters were $\eta = 1 \times 10^{-2}$, $\beta = 0$ and $E = 5$ and the instantiation, training and testing of the two networks were repeated 15 times independently. Non-linear activations were ReLU and the MSE loss was minimized.

III. Results

The code of all the modules described in Section II, as well as convenient functions for training the network and computing the number of classification errors, are provided in the file `nn_group14.py`. The PyTorch model is provided by the routines in `nn_pytorch.py` and the naive implementation in `nn_group14_naive.py`. Experiments I through IV were run with the script `accuracy_study.py` and Experiments V and VI with `performance_study.py`.

A. End-user experience

Listing 1 provides the end-user code for handy creation, training and testing of a neural network. Our end-user function calls are quite similar to PyTorch in that `Module.backward`, `Loss.backward` and `Optimizer.step` essentially perform most of the training. The `Sequential` module allows the user to combine layers arbitrarily, including `Sequential` modules themselves.

Listing 1: General code for training a neural network using our framework.

```
import nn_group14 as nn14
# Create model
non_lin_activ = nn14.ReLU  # ReLU, Tanh, Sigmoid
nonlin = non_lin_activ.nonlin_str()

fc1 = nn14.Linear(2, 25, nonlinearity=nonlin)
fc2 = nn14.Linear(25, 25, nonlinearity=nonlin)
fc3 = nn14.Linear(25, 25, nonlinearity=nonlin)
fc_out = nn14.Linear(25, 2)
model = nn14.Sequential([fc1, non_lin_activ(),
                        fc2, non_lin_activ(),
                        fc3, non_lin_activ(),
                        fc_out])

# Loss function: MSELoss(), NLLLoss(), CrossEntropyLoss()
criterion = nn14.MSELoss()

# Optimizer
optimizer = nn14.SGD(model.param(), lr=1e-2, momentum=0.85)

# Training
for i_ep in range(n_epochs):
    for b_start in range(0, Ntrain, batch_size):
        model.zero_grad()
        # account for boundary effects
```

```
bsize_eff = batch_size - max(0, b_start + batch_size - Ntrain)
# forward pass
output = model.forward(train_input.narrow(0, b_start, bsize_eff))
# backward pass
dl_dout = criterion.backward(output, train_target.narrow(0, b_start,
        bsize_eff))
dl_dx = model.backward(dl_dout)
# parameter update
optimizer.step()
```

B. Validation on a toy example

Experiment I. Effect of momentum on classification error: Figures 6a and 6b suggest that our framework compared favorably to PyTorch. The test error was systematically above the train error as expected. The results also point to the relative instability of neural networks: for low values of β, the PyTorch network did not converge although its architecture and training were very similar to ours, essentially differing in the way parameters were initialized. In this particular case, the MSE loss yielded lower classification errors.

Experiment II. Effect of learning rate on classification error: Figures 7a and 7b show that convergence and optimal classification occurred at lower learning rates with ReLU than with Tanh, which did not reach an optimum in the range of values considered in our experiment. The performance of our framework were globally similar to those of PyTorch although for learning rates differing by about an order of magnitude.

Experiment III. Effect of number of epochs on classification error: Figures 8a and 8b show that the train and test errors generally decreased as the number of epochs increased. Our framework obtained errors that were equal to or lower than those of PyTorch and convergence was slightly slower with CE loss than with MSE loss.

Experiment IV. Effect of parameter initialization on classification error: Figure 9 shows that the results were very similar for our network and its PyTorch homologue across all initialization strategies. The results indicate that proper parameter initialization is critical to the performance of the network, with test error rates falling from about 50% for $a = 0.2$ down to about 2% with $a = 1.0$.

Experiment V. Effect of batch size on computation time: Figures 10a and 10b show that the computation time globally decreased as the batch size increased, with particularly long computation times for very small batch sizes. The trend seemed to break around $B \approx 500$ with our framework, whereas it remained steady for PyTorch in the range of values examined in our experiment. This may be explained by a better use of memory and a better handling of memory transfers in PyTorch.

Experiment VI. Effect of vectorizing tensor operations on computation time: Vectorizing tensor operations using libraries like PyTorch `Tensor` lead to dramatic gains in computation times: the naive version of the code executed in $91.1 \pm 1.79\,\mathrm{s}$ while the `Tensor`-based implementation ran in $0.71 \pm 0.0019\,\mathrm{s}$, i.e. a speedup of about 128. As observed in Experiment V, the speedup can be even greater when using batch sizes $B > 1$.

IV. Discussion and conclusion

The results obtained with our framework proved globally comparable to those of PyTorch for all the features that we implemented, performing even slightly better in Experiments I, II (with Tanh) and III. However, one should refrain from labeling one implementation as better or worse due to the relatively high sensitivity of the results to initialization or to optimization meta-parameters. In Experiment II for instance, we found that the optimal learning rates were very different for the two frameworks. Experiment IV suggested that the main source of difference between the two implementations was the parameter initialization, as the results were much more similar when an identical initialization strategy was enforced. This may in turn have impacted the results of Experiments I and III, which considered identical learning rates but different initialization strategies. Due to time constraints, we were not able to assess the effect of noise on the labels on our overall performance.

Relatively direct extensions of the presented framework include adapting the learning rate to the rate of change of the loss. Xavier's initialization to control both the variance of the activations *and* the variance of the gradient of the loss with respect to the activations would also be straightforward to include. Our code lends itself to transferring all the `Tensor` contained in the model to be trained to a GPU via the `Module.param` method. Implementing dropout would likely involve a reasonable amount of effort as well. More involved improvements include batch norm layers due to the complexity of their derivatives and interdependence between batch samples. L2 regularization of the input may require some work given that the loss function would then depend on the network input as well. Adding more sophisticated features such as recurrent or convolutional layers would require a more powerful code design such as in `torch.autograd`, which was outside the scope of this project.

In conclusion, we have been able to implement neural networks with an MLP-like architecture, coding the forward and backward pass operations efficiently with `Tensor` operations. Various experiments suggest that the network performs as expected on a toy problem of binary classification and that the code readily lends itself to further refinements.

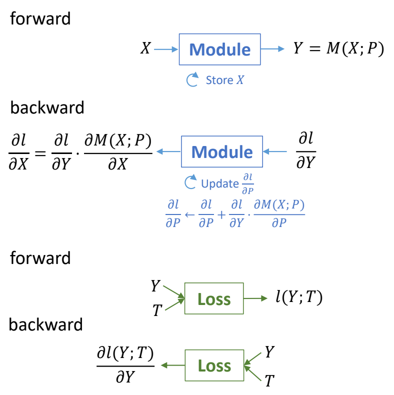
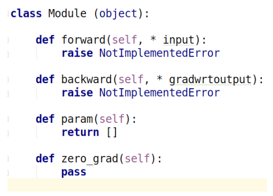
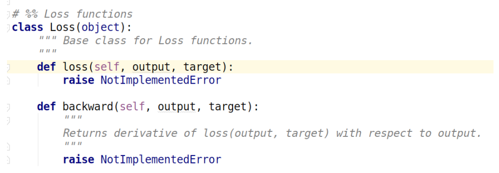

forward
X
Module
$Y = M(X;P)$
Store X
backward
$\frac{\partial l}{\partial X} = \frac{\partial l}{\partial Y} \cdot \frac{\partial M(X;P)}{\partial X}$
Module
$\frac{\partial l}{\partial Y}$
Update $\frac{\partial l}{\partial P}$
$\frac{\partial l}{\partial P} \leftarrow \frac{\partial l}{\partial P} + \frac{\partial l}{\partial Y} \cdot \frac{\partial M(X;P)}{\partial P}$
forward
Y
T
Loss
$l(Y;T)$
backward
$\frac{\partial l(Y;T)}{\partial Y}$
Loss
Y
T

Fig. 2: Schematic illustration of the forward and backward pass of our `Module` and `Loss` objects. Input is X, output is Y and the output target is T.



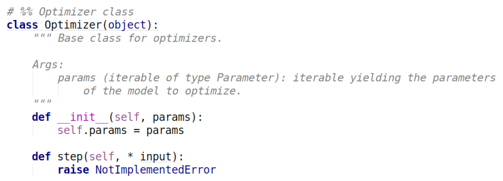
```
class Module (object):

    def forward(self, * input):
        raise NotImplementedError

    def backward(self, * gradwrtoutput):
        raise NotImplementedError

    def param(self):
        return []

    def zero_grad(self):
        pass
```

Fig. 3: `Module` base class.



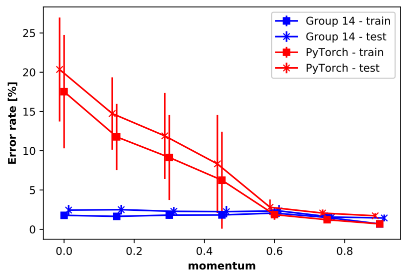
```
# %% Loss functions
class Loss(object):
    """ Base class for Loss functions.
    """
    def loss(self, output, target):
        raise NotImplementedError

    def backward(self, output, target):
        """
        Returns derivative of loss(output, target) with respect to output.
        """
        raise NotImplementedError
```

Fig. 4: `Loss` base class.

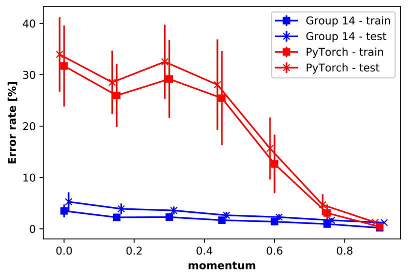


```
# %% Optimizer class
class Optimizer(object):
    """ Base class for optimizers.

    Args:
        params (iterable of type Parameter): iterable yielding the parameters
            of the model to optimize.
    """
    def __init__(self, params):
        self.params = params

    def step(self, * input):
        raise NotImplementedError
```

Fig. 5: `Optimizer` base class.

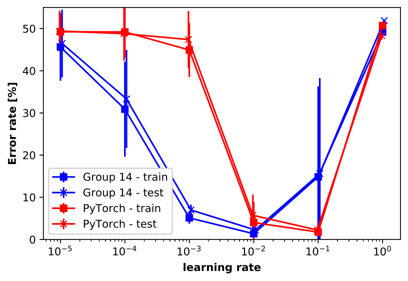


(a) Using the MSE loss.

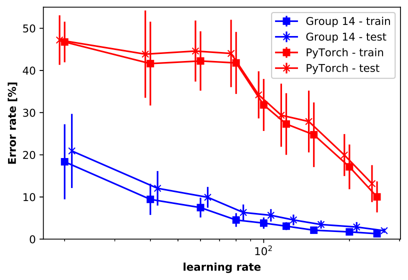


(b) Using the cross-entropy loss.

Fig. 6: (Exp. I) The error rates of our method were equal to or lower than those of PyTorch. Momentum can have a significant impact on performance. The x values are slightly offset for visualization purposes.

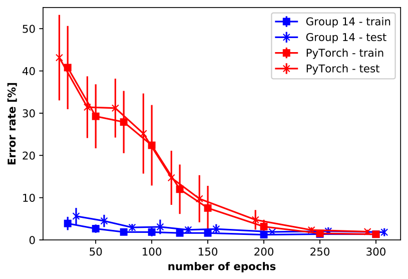


(a) Using ReLU activations.

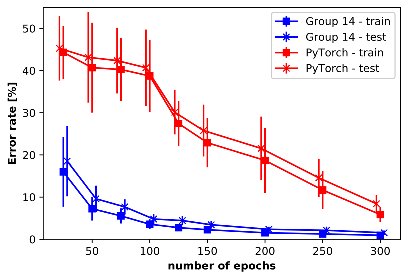


(b) Using Tanh activations.

Fig. 7: (Exp. II) Different dynamics with respect to η were observed using ReLU or Tanh activations. The two implementations feature similar performance although at different values of the learning rate. The x values are slightly offset for visualization purposes.



(a) Using the MSE loss.



(b) Using the cross-entropy loss.

Fig. 8: (Exp. III) More epochs lead to lower train and test errors. The x values are slightly offset for visualization purposes.

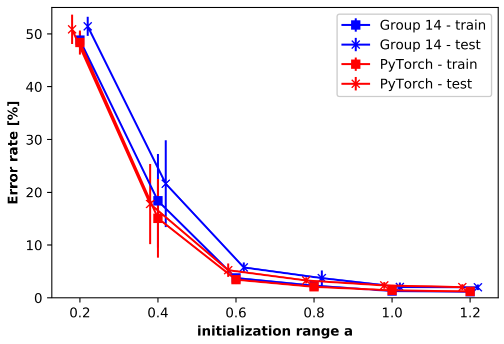
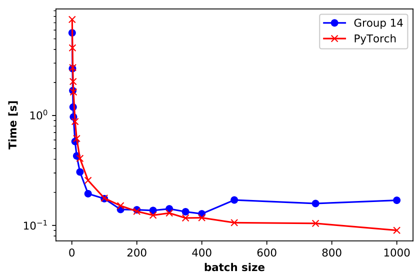
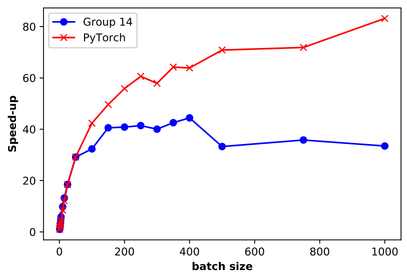


Fig. 9: (Exp. IV) Classification performance is heavily affected by the the range a of parameters during initialization. With similar initialization strategies, the results of our framework were very similar to those of PyTorch.



(a) Logarithm of mean execution time vs batch size.



(b) Speed-up (computed with respect to batch size $B = 1$) vs batch size.

Fig. 10: (Exp. V) Results suggest similar performance for our framework compared to PyTorch up until a batch size of about 500, where memory overhead may start impacting our performance.

References

[1] Darrell Etherington. Waymo racks up 4 million self-driving miles. https://techcrunch.com/2017/11/27/waymo-racks-up-4-million-self-driven-miles/, Nov 2017. [Online; accessed May-18-2018].

[2] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision*, pages 1026–1034, 2015.

[3] Brody Huval, Tao Wang, Sameep Tandon, Jeff Kiske, Will Song, Joel Pazhayampallil, Mykhaylo Andriluka, Pranav Rajpurkar, Toki Migimatsu, Royce Cheng-Yue, et al. An empirical evaluation of deep learning on highway driving. *arXiv preprint arXiv:1504.01716*, 2015.

[4] Eric Mack. New MIT System Lets Self-Driving Cars Go Almost Anywhere. https://www.forbes.com/sites/ericmack/2018/05/07/mit-ai-let-self-driving-cars-go-anywhere-prevent-uber-accident/#4e9171e929c2, May 2018. [Online; accessed May-17-2018].

[5] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *nature*, 323(6088):533, 1986.

[6] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. Mastering the game of go without human knowledge. *Nature*, 550(7676):354, 2017.