Università degli Studi di Napoli Federico II

Dipartimento di Ingegneria Elettrica e delle Tecnologie dell'Informazione

Neuro BackPropagation Lab

Giuliano Aiello

2025

Contents

Chapter 1

Prolusion

1.1 Goal

This report provides a comprehensive overview of a Python project whose goal is to develop and compare different adaptive backpropagation techniques involved in a machine learning process, as Rprop (Resilient BackPropagation). MNIST is the target of the learning model.

The project follows the "Empirical evaluation of the improved Rprop learning algorithms" article by Christian Igel and Michel Hüsken (2001).

1.2 Software Stack

- Python 3.9.6
- PyTorch 2.6.0

The project is equipped with a `requirements.txt` file which allows for seamless installation of dependencies, by executing `pip install -r requirements.txt`.

1.3 Project Structure

```
neuro-backprop-lab/
├── model/
├── tester/
│   ├── tester.py
│   └── <trained_model>.pt
├── trainer/
│   ├── irpropplus/
│   ├── rpropminus/
│   ├── rpropplus/
│   └── trainer.py
├── utils/
│   └── loader_dataset.py
├── test_model.py
└── train_model.py
```

- `model` includes the neural network model architecture.
- `tester` handles the testing flow of the ready-to-use <trained_model>.pt.
- `trainer` handles the examined backpropagation techniques and the training flow of the model, saving it as <trained_model>.pt.
- `utils` offers utility functions designed to support the root project scripts.

Chapter 2

Module Overview

The part of the project which is shared across all the examined Rprop techniques is presented as follows.

2.1 Train Model

This script runs the training flow of the model and saves the model for future testing phase.

Algorithm 1: `train_model.py`

$model \leftarrow newModel()$
$criterion, optimizer, epochs, train\_set, eval\_set \leftarrow get\_configuration()$

$Trainer.traineval(model, criterion, optimizer, train\_set, eval\_set, epochs)$

2.2 Test Model

This script runs the test flow of the model.

Algorithm 2: `test_model.py`

$model, optimizer \leftarrow load\_model()$
$criterion, test\_set \leftarrow get\_configuration()$

$Tester.test(model, criterion, test\_set)$

2.3 Model

This class, `model.py`, represents the artificial neural network model architecture to be trained and tested. What follows are empirical choices that are the result of various experiments.

The model is a shallow network based on `torch.nn.Module`[1] class. Its three layers are fully connected using `torch.nn.Linear(.)` and they feed forward as follows:

1. the first layer flattens the input MNIST image, by transforming it from a multidimensional vector to a 784-sized (since a 28×28-sized image is manipulated) one-dimensional vector;
2. the hidden layer receives the transformed vector and processes it into a 128-sized vector with a ReLU activation function to introduce non-linearity;
3. the output layer extracts the final predictions by transforming the 128-sized vector into a 10-sized vector, which corresponds to the number of possible classes for classification.

[1] *https://pytorch.org/docs/stable/generated/torch.nn.Module.html* (accessed 2025)

2.4 Trainer

This class, `trainer.py`, is responsible for training the model and saving it for future testing phase. Data retrieval is addressed in subsection 2.6.1.

Algorithm 3: `trainer.py`

```
Function traineval(model, criterion, optimizer, train_set, eval_set, epochs):
    foreach epoch ∈ epochs do
        train_loss_avg, train_accuracy ←
          train(model, criterion, optimizer, train_set, train_loss_avgs, train_accuracies)
        eval_loss_avg, eval_accuracy ← eval(model, criterion, eval_set, eval_loss_avgs, eval_accuracies)
        if eval_loss_avg < eval_loss_avg_prev then
            savemodel(model)
        end
    end
    return train_loss_avgs, train_accuracies, eval_loss_avgs, eval_accuracies
return

Function train(model, criterion, optimizer, train_set, loss_averages, accuracies):
    foreach batch ∈ train_set do
        labels, loss, outputs ← trainstep(model, criterion, optimizer, batch)
        total_correct, total_loss, total_samples ←
          gather_metrics(labels, loss, outputs, total_correct, total_loss, total_samples)
    end
    loss_average, accuracy ←
      compute_metrics(total_correct, total_loss, total_samples, loss_averages, loss_accuracies)
    return loss_average, accuracy
return
Function trainstep(model, criterion, optimizer, batch):
    inputs, labels ← batch
    outputs ← model(inputs)
    loss ← criterion(outputs, labels)
    loss.compute_gradients()
    optimizer.step()
    return labels, loss, outputs
return
Function eval(model, criterion, eval_set, loss_averages, accuracies):
    foreach batch ∈ eval_set do
        labels, loss, outputs ← evalstep(model, criterion, batch)
        total_correct, total_loss, total_samples ←
          gather_metrics(labels, loss, outputs, total_correct, total_loss, total_samples)
    end
    loss_average, accuracy ←
      compute_metrics(total_correct, total_loss, total_samples, loss_averages, accuracies)
    return loss_average, accuracy
return
Function evalstep(model, criterion, batch):
    inputs, labels ← batch
    outputs ← model(inputs)
    loss ← criterion(outputs, labels)
    return labels, loss, outputs
return
```

2.5 Tester

This class, `tester.py`, is responsible for loading and testing a pre-trained model on unseen data.

Algorithm 4: `tester.py`

Input : model, criterion, test_set

Function *test*(*model, criterion, test_set*):
    foreach *batch* $\in$ *test_set* **do**
        $labels, loss, outputs \leftarrow eval(model, criterion, batch)$
        $total\_correct, total\_loss, total\_samples \leftarrow gather\_metrics(labels, loss, outputs)$
    end
    $loss\_average, accuracy \leftarrow compute\_metrics(total\_correct, total\_loss, total\_samples)$
return
Function *eval*(*model, criterion, batch*):
    $inputs$, $labels \leftarrow batch$
    $outputs \leftarrow model(inputs)$
    $loss \leftarrow criterion(outputs, labels)$

    return *labels*, *loss*, *outputs*
return

2.6 Utils

2.6.1 Loader Dataset

This class, `loader-dataset.py`, is responsible for the methodology employed for data retrieval. It is the module that takes the most importance of the `utils` package, so it was worth describing it.

The method used to get the data is holdout, there is more than one function involved in this.

Algorithm 5: `loader_dataset.py`

Function *getdatasettraineval*(*dataset_size_train, batch_size_train, dataset_size_eval, batch_size_eval*)**:**
  $learning\_set \leftarrow get\_MNIST\_dataset(learn\_mode = True)$
  $dataset\_train, dataset\_eval \leftarrow$
   $split(learning\_set, dataset\_size\_train, batch\_size\_train, dataset\_size\_eval, batch\_size\_eval)$
  return $dataset\_train, dataset\_eval$
return

Function *getdatasettest*(*dataset_size, batch_size*)**:**
  $dataset\_test \leftarrow get\_MNIST\_dataset(learn\_mode = False)$
  return $dataset\_test(batch\_size)$
return

As planned, the learning data is completely separated from the testing data with the `learn_mode` parameter.

The learning set is, in turn, split into the train set for the training phase and in the eval set for the evaluation phase. Finally, the train set undergoes a data shuffle, this should make the train phase more robust. The eval set comes in handy to avoid overfitting.

It is also important to note that the MNIST allocates $60,000$ samples for training and $10,000$ for testing.

Chapter 3

Resilient Backpropagation Techniques

Rprop algorithms differ from the classical back-propagation algorithms by the fact that they are independent of the magnitude of the gradient, but depend on its sign only.

3.1 Implementations

Recall that the main concern of the documentation is readability. Hence, pseudocode and actual code implementations may slightly differ, as the Python scripting language allows for significant performance improvements through the use of native structures. These differences clearly don't affect the functionality of the implementations.

Each Rprop algorithm that will be described corresponds to a specialized `torch.optim.Optimizer.step()`[1] class method.

An Rprop algorithm is intended to perform the following steps:

1. Compute the gradient of the error function with respect to the model weights.
2. Update the step size based on a conditional logic of the current and previous gradient sign:

$$\Delta_{ij}^{curr} = \begin{cases} \min(\eta^+ \cdot \Delta_{ij}^{prev}, \Delta_{\max}) & \text{if} \frac{\partial E}{\partial w_{ij}}^{curr} \cdot \frac{\partial E}{\partial w_{ij}}^{prev} > 0 \\ \max(\eta^- \cdot \Delta_{ij}^{prev}, \Delta_{\min}) & \text{if} \frac{\partial E}{\partial w_{ij}}^{curr} \cdot \frac{\partial E}{\partial w_{ij}}^{prev} < 0 \\ \Delta_{ij}^{prev} & \text{otherwise} \end{cases}$$

3. Update the step size direction using either weight-backtracking or the gradient sign.
4. Update weights with the step size direction.

Subsequently, each variant of the algorithm implements its own adaptation of this general process.

[1] *https://pytorch.org/docs/main/optim.html* (accessed 2025)

3.1.1 Rprop without Weight-Backtracking

This Rprop version, also said Rprop-, updates the step size direction with the gradient sign only.

Even though the gradient-product case resulting in zero is a no-op, it is kept for readability and consistency with other algorithms.

Algorithm 6: `RpropMinus.step()`

for *parameter* $\in$ *parameters* **do**

  if $\frac{\partial E}{\partial w_{ij}}^{curr} \cdot \frac{\partial E}{\partial w_{ij}}^{prev} > 0$ **then**

    $\Delta_{ij}^{curr} = \min(\eta^{+} \cdot \Delta_{ij}^{prev}, \Delta_{\max})$

  end

  if $\frac{\partial E}{\partial w_{ij}}^{curr} \cdot \frac{\partial E}{\partial w_{ij}}^{prev} < 0$ **then**

    $\Delta_{ij}^{curr} = \max(\eta^{-} \cdot \Delta_{ij}^{prev}, \Delta_{\min})$

  end

  if $\frac{\partial E}{\partial w_{ij}}^{curr} \cdot \frac{\partial E}{\partial w_{ij}}^{prev} = 0$ **then**

    $\Delta_{ij}^{curr} = \Delta_{ij}^{prev}$

  end

  $\Delta w_{ij}^{curr} = -\operatorname{sign}(\frac{\partial E}{\partial w_{ij}}^{curr}) \cdot \Delta_{ij}^{curr}$

  $w_{ij}^{curr} = w_{ij}^{prev} + \Delta w_{ij}^{curr}$

end

3.1.2 Rprop with Weight-Backtracking

This Rprop version, also said Rprop^{+}, updates the step size direction with the gradient sign when the gradient product is greater than or equal to zero. In the other case the algorithm performs a weight-backtracking, formally $\Delta w_{ij}^{curr} = -\Delta w_{ij}^{prev}$, then the current gradient is set to zero in order to activate the gradient-product case resulting in zero in the next iteration.

Algorithm 7: `RpropPlus.step()`

---

for *parameter* $\in$ *parameters* **do**
 if $\frac{\partial E}{\partial w_{ij}}^{curr} \cdot \frac{\partial E}{\partial w_{ij}}^{prev} > 0$ **then**
  $\Delta_{ij}^{curr} = \min(\eta^{+} \cdot \Delta_{ij}^{prev}, \Delta_{\max})$
  $\Delta w_{ij}^{curr} = -\operatorname{sign}(\frac{\partial E}{\partial w_{ij}}^{curr}) \cdot \Delta_{ij}^{curr}$
 end
 if $\frac{\partial E}{\partial w_{ij}}^{curr} \cdot \frac{\partial E}{\partial w_{ij}}^{prev} < 0$ **then**
  $\Delta_{ij}^{curr} = \max(\eta^{-} \cdot \Delta_{ij}^{prev}, \Delta_{\min})$
  $\Delta w_{ij}^{curr} = -\Delta w_{ij}^{prev}$
  $\frac{\partial E}{\partial w_{ij}}^{curr} = 0$
 end
 if $\frac{\partial E}{\partial w_{ij}}^{curr} \cdot \frac{\partial E}{\partial w_{ij}}^{prev} = 0$ **then**
  $\Delta_{ij}^{curr} = \Delta_{ij}^{prev}$
  $\Delta w_{ij}^{curr} = -\operatorname{sign}(\frac{\partial E}{\partial w_{ij}}^{curr}) \cdot \Delta_{ij}^{curr}$
 end
 $w_{ij}^{curr} = w_{ij}^{prev} + \Delta w_{ij}^{curr}$
end

3.1.3 Improved Rprop with Weight-Backtracking

This Rprop version, also said IRprop$^+$, extends the one described in subsection 3.1.2.

The only modification concerns the gradient-product case resulting in less than zero: weight-backtracking is adopted only if the current error is greater than the previous error (this is what is meant by 'improved'), otherwise the step size direction is set to zero.

Algorithm 8: `IRpropPlus.step()`

for $parameter \in parameters$ **do**
  if $\frac{\partial E}{\partial w_{ij}}^{curr} \cdot \frac{\partial E}{\partial w_{ij}}^{prev} > 0$ **then**
    $\Delta_{ij}^{curr} = \min(\eta^+ \cdot \Delta_{ij}^{prev}, \Delta_{\max})$
    $\Delta w_{ij}^{curr} = -\operatorname{sign}(\frac{\partial E}{\partial w_{ij}}^{curr}) \cdot \Delta_{ij}^{curr}$
  end
  if $\frac{\partial E}{\partial w_{ij}}^{curr} \cdot \frac{\partial E}{\partial w_{ij}}^{prev} < 0$ **then**
    $\Delta_{ij}^{curr} = \max(\eta^- \cdot \Delta_{ij}^{prev}, \Delta_{\min})$
    if $E^{curr} > E^{prev}$ **then**
      $\Delta w_{ij}^{curr} = -\Delta w_{ij}^{prev}$
    else
      $\Delta w_{ij}^{curr} = 0$
    end
    $\frac{\partial E}{\partial w_{ij}}^{curr} = 0$
  end
  if $\frac{\partial E}{\partial w_{ij}}^{curr} \cdot \frac{\partial E}{\partial w_{ij}}^{prev} = 0$ **then**
    $\Delta_{ij}^{curr} = \Delta_{ij}^{prev}$
    $\Delta w_{ij}^{curr} = -\operatorname{sign}(\frac{\partial E}{\partial w_{ij}}^{curr}) \cdot \Delta_{ij}^{curr}$
  end
  $w_{ij}^{curr} = w_{ij}^{prev} + \Delta w_{ij}^{curr}$
end

3.1.4 Rprop with Weight-Backtracking by PyTorch

Additionally, the PyTorch version of Rprop$^+$ is provided as an extra implementation[2].

Thus, as said in sec. 3.1, this is the uncustomized `Optimizer.step()` method.

[2] *https://pytorch.org/docs/stable/generated/torch.optim.Rprop.html* (accessed 2025)

3.2 Comparisons

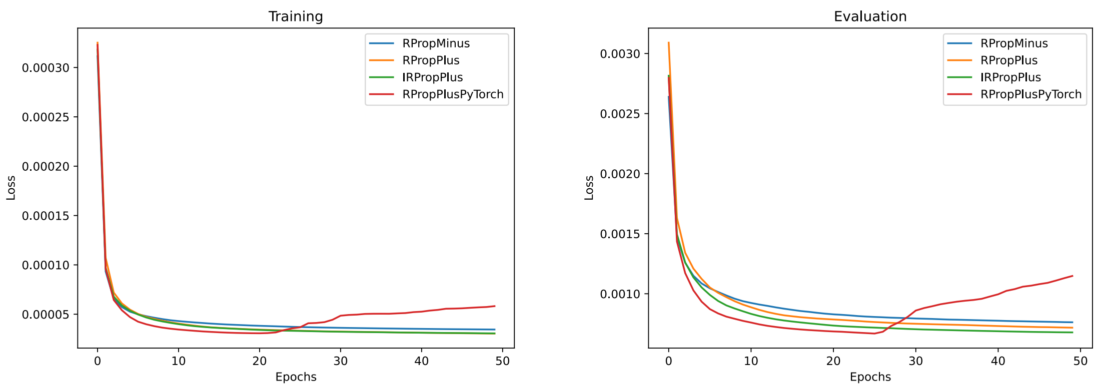
The four distinct techniques proposed for Rprop are compared through a structured experimental workflow.

Each technique is measured in terms of loss and accuracy. The loss function adopted is the cross-entropy loss:

$$\sum_{n=1}^{N} - \sum_{k=1}^{c} t_k^n \ln y_k^n$$

A total of $60,000$ elements was instantiated for the dataset, with $50,000$ allocated for the training set and $10,000$ for the evaluation set, using batch sizes of $5,000$ and 250, respectively. The learning rate is set as 0.001, the procedure was performed over 50 epochs. A coherent configuration file varying on a specific Rprop version is:

```
"criterion": "cross_entropy",
"optimizer": <rproptechnique>,
"learning_rate": 0.001,
"epochs": 50,
"train_set_size": 50000,
"train_batch_size": 5000,
"eval_set_size": 10000,
"eval_batch_size": 250
```



It can be instantly observed that the three different customized versions of Rprop exhibit the same trend during the training phase.

Acronyms

IRprop Improved Resilient BackPropagation 16

MNIST Modified National Institute of Standards and Technology database 1, 8, 12

ReLU Rectified Linear Unit 8

Rprop Resilient BackPropagation 1, 5, 13–18