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). Target of the learning model is the MNIST.

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/
├── 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 it for future testing phase.

Algorithm 1: `train_model.py`

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

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

$savemodel(model)$

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.

It 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, a choice that was the result of empirical experiments;
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.

Algorithm 3: `trainer.py`

Function *traineval*(*model, criterion, optimizer, train_set, eval_set, epochs*)**:**
  foreach $epoch \in epochs$ **do**
    $train\_loss\_avg, train\_accuracy \leftarrow$
      $train(model, criterion, optimizer, train\_set, train\_loss\_avgs, train\_accuracies)$
    $eval\_loss\_avg, eval\_accuracy \leftarrow eval(model, criterion, eval\_set, eval\_loss\_avgs, eval\_accuracies)$
  end
  return $train\_loss\_avgs, train\_accuracies, eval\_loss\_avgs, eval\_accuracies$
return

Function *train*(*model, criterion, train_set, loss_averages, accuracies*)**:**
  foreach $batch \in train\_set$ **do**
    $labels, loss, outputs \leftarrow trainstep(model, criterion, optimizer, batch)$
    $total\_correct, total\_loss, total\_samples \leftarrow$
      $gather\_metrics(labels, loss, outputs, total\_correct, total\_loss, total\_samples)$
  end
  $loss\_average, accuracy \leftarrow$
    $compute\_metrics(total\_correct, total\_loss, total\_samples, loss\_averages, loss\_accuracies)$
  return $loss\_average, accuracy$
return
Function *trainstep*(*model, criterion, optimizer, batch*)**:**
  $inputs, labels \leftarrow batch$
  $outputs \leftarrow model(inputs)$
  $loss \leftarrow criterion(outputs, labels)$
  $loss.compute\_gradients()$
  $optimizer.step()$
  return $labels, loss, outputs$
return
Function *eval*(*model, criterion, eval_set, loss_averages, accuracies*)**:**
  foreach $batch \in eval\_set$ **do**
    $labels, loss, outputs \leftarrow evalstep(model, criterion, batch)$
    $total\_correct, total\_loss, total\_samples \leftarrow$
      $gather\_metrics(labels, loss, outputs, total\_correct, total\_loss, total\_samples)$
  end
  $loss\_average, accuracy \leftarrow$
    $compute\_metrics(total\_correct, total\_loss, total\_samples, loss\_averages, accuracies)$
  return *loss_average, accuracy*
return
Function *evalstep*(*model, criterion, batch*)**:**
  $inputs, labels \leftarrow batch$
  $outputs \leftarrow model(inputs)$
  $loss \leftarrow criterion(outputs, labels)$
  return $labels, loss, outputs$
return

2.5 Tester

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

Algorithm 4: `tester.py`

Input : model, criterion, test_set

Function *test*(*model, criterion, batch*)**:**
- **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.

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

3.1.1 Rprop-

This is Rprop-.

3.1.2 Rprop+

This is Rprop+.

3.1.3 Improved Rprop with Weight-Backtracking

This is IRprop+.

3.1.4 Rprop+ by PyTorch

This is Rprop+ by PyTorch.[1]

3.2 Comparisons

Here I will show comparisons.

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

Acronyms

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

ReLU Rectified Linear Unit 8

Rprop Resilient BackPropagation 1, 5, 13