


Free Project Techhnical Report - M2 CompuPhys

- Technical Report - Implementation of Direct Feedback Alignment Using PyTorch

Léo BECHET
University of Franche-Comté

Under the supervision of Pr.Daniel BRUNNER,
and Dr.Anas SKALLI from the FEMTO-ST Institute

Contents

1 Introduction **2**

2 Alpha Version : Proof Of Concept **2**

3 Beta Version: Framework **5**
3.1 User guide . . . 5
3.2 Model Class . . . 5
3.2.1 Initialization . . . 5
3.2.2 Forward Passes . . . 6
3.2.3 DFA Backward Pass . . . 7
3.2.4 Printing a Summary of the Model . . . 7
3.3 Training loops . . . 8
3.3.1 Basic training loop . . . 8
3.3.2 Batch-averaged training loop . . . 10
3.3.3 Per class batch-averaged training loop . . . 11

4 How to use the provided demos **12**

5 Free Project versioning breakdown **12**
5.0.1 Alpha . . . 12
5.0.2 Beta . . . 12
5.0.3 Gold . . . 13

Abstract

The following projects aims to provide a basic framework which can be used to train Artificial Neural Networks using Direct Feedback Alignment. We use the popular PyTorch package to provide a robust and already well documented base framework.

Keywords: Python, PyTorch, Direct Feedback Alignment, Backpropagation, Machine Learning, Neural Network Training.

1 Introduction

In this project, we aim to create a framework for training PyTorch networks using Direct Feedback Alignment (DFA). To begin with, we have developed an Alpha version of the framework, which serves as a proof of concept and interactive tool. The Alpha version is designed in a Jupyter notebook format, allowing users to explore different aspects of the algorithm.

Our approach with the Alpha version is to familiarize ourselves with the principles of DFA training using a fixed network size, before diving into the actual implementation of the framework. This initial stage helps us understand the underlying concepts better and enables us to make informed decisions while building the final product.

Following the success of the Alpha version, we have moved on to create a Beta version that adheres to the different approaches outlined in our scientific report. The primary objective of this framework is to provide users with a solid foundation upon which they can build their DFA-based projects.

In conclusion, both the Alpha and Beta versions of our Direct Feedback Alignment framework are crucial steps towards empowering users to train PyTorch networks using DFA.

2 Alpha Version : Proof Of Concept

This section presents the proof-of-concept implementation of our approach using a Jupyter notebook. Each component of the implementation is modular and partly-documented. We start by importing the required dependencies for the project. These include libraries such as `torch` for deep learning, `numpy` for numerical computations, and `scipy.special` for activation functions. The imports are as follows:

```
import torch
import copy
import torch.nn as nn
from torch.utils.data import TensorDataset, DataLoader
import torch.nn.functional as F
import numpy as np
from scipy.special import expit
```

We define a simple artificial neural network (ANN) model using PyTorch, with flexibility to specify the layer sizes and activation function. The model's forward pass propagates input through each layer while applying the activation function to all but the last layer. The implementation is as follows:

```
class DynamicModel(nn.Module):
    def __init__(self, layer_sizes, act_function):
        super(DynamicModel, self).__init__()
        self.layers = nn.ModuleList(
            [nn.Linear(layer_sizes[i], layer_sizes[i+1]) for i in range(len(layer_sizes) - 1)]
        )
        self.act_function = act_function
```

```
    def forward(self, x):
        for i, layer in enumerate(self.layers):
            x = layer(x)
            if i < len(self.layers) - 1:
                x = self.act_function(x)
        return x
```

To extend functionality, we define a forward pass that outputs intermediate activations in addition to the final predictions. This is crucial for implementing a Direct Feedback Alignment (DFA) backward pass. The forward pass returns the intermediate activations a_1, h_1, a_2, and predictions y_{hat}. lease note that we hardcoded the return values. This is due to the fact that we will only use 1 hidden layer in this example.

```
def forward_pass(model, x):
    h1 = model.act_function(model.layers[0](x))
    a1 = model.layers[0](x)
    a2 = model.layers[1](h1)
    y_hat = torch.sigmoid(a2)
    return a1, h1, a2, y_hat
```

The DFA backward pass computes weight updates using a random feedback matrix and activation derivatives. This avoids reliance on backpropagation by propagating the error directly through the feedback paths. Again, please note that weights and bias updates are hardcoded for a 1 hidden layer network.

```
def dfa_backward_pass(e, h1, B1, a1, x):
    dW2 = -torch.matmul(e, h1.T)
    da1 = torch.matmul(B1, e) * (1 - torch.tanh(a1)**2)
    dW1 = -torch.matmul(da1, x.T)
    db1 = -torch.sum(da1, dim=1, keepdim=True)
    db2 = -torch.sum(e, dim=1, keepdim=True)
    return dW1, dW2, db1, db2
```

A custom training loop was implemented for training the model using DFA. The loop involves calculating loss, performing the forward pass, and manually updating weights based on the gradients computed through DFA. Random feedback matrices are initialized for DFA computation. Training error and loss are monitored across epochs. Early stopping is triggered based on a tolerance parameter to ensure computational efficiency:

```
def train_DFA(model, x, y, n_epochs=10, lr=1e-3, batch_size=200, tol=1e-1):
    # Initialization and dataset setup
    ...
    for epoch in range(n_epochs):
        for batch_x, batch_y in dataloader:
            # Forward pass
            a1, h1, a2, y_hat = forward_pass(model, batch_x)
            error = y_hat - batch_y

            # Compute gradients and update weights
            dW1, dW2, db1, db2 = dfa_backward_pass(error.T, h1.T, B1, a1.T, batch_x.T)
            with torch.no_grad():
                model.layers[0].weight += lr * dW1
                model.layers[0].bias += lr * db1.squeeze()
                model.layers[1].weight += lr * dW2
                model.layers[1].bias += lr * db2.squeeze()
        ...
```

To evaluate the proof of concept, the MNIST dataset is loaded and preprocessed. Input features are normalized, and labels are one encoded for classification. The model architecture includes 784 input neurons (28×28 input images loaded linearly), 200 hidden neurons in 1 layer, and 10 output neurons(0-9 digits). We trained two models: one using DFA and another using standard backpropagation. Both models were trained for 80 epochs with a batch size of 200 and a learning rate of 10^{-4}.

```
# Load and preprocess MNIST
from keras.datasets import mnist
(X_train, y_train), (X_test, y_test) = mnist.load_data()
...
y_train = np.eye(nb_classes)[y_train]
y_test = np.eye(nb_classes)[y_test]
```

Finally, training errors for both models were plotted, revealing the expected behavior of decreasing error rates. Results on the test dataset showed comparable performance for DFA and backpropagation, with around 97% accuracy across all digits. Confusion matrices, precision, recall, and other metrics confirm that DFA provides consistent results while maintaining the simplicity of its approach. Fig.1 illustrates the training error evolution and performance metrics are displayed in Fig.2.

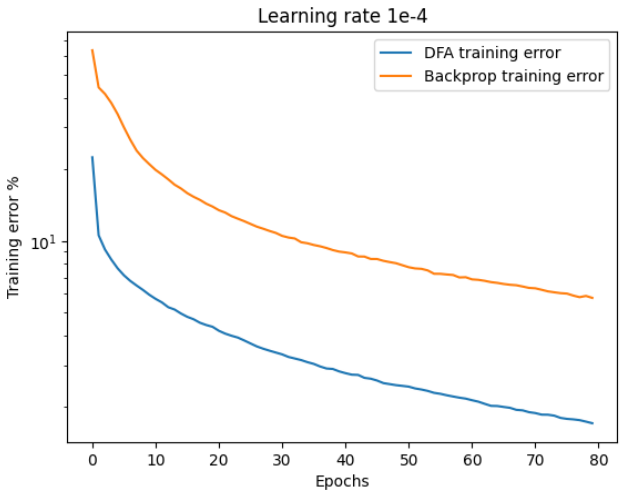

Figure 1: Training error on an example run

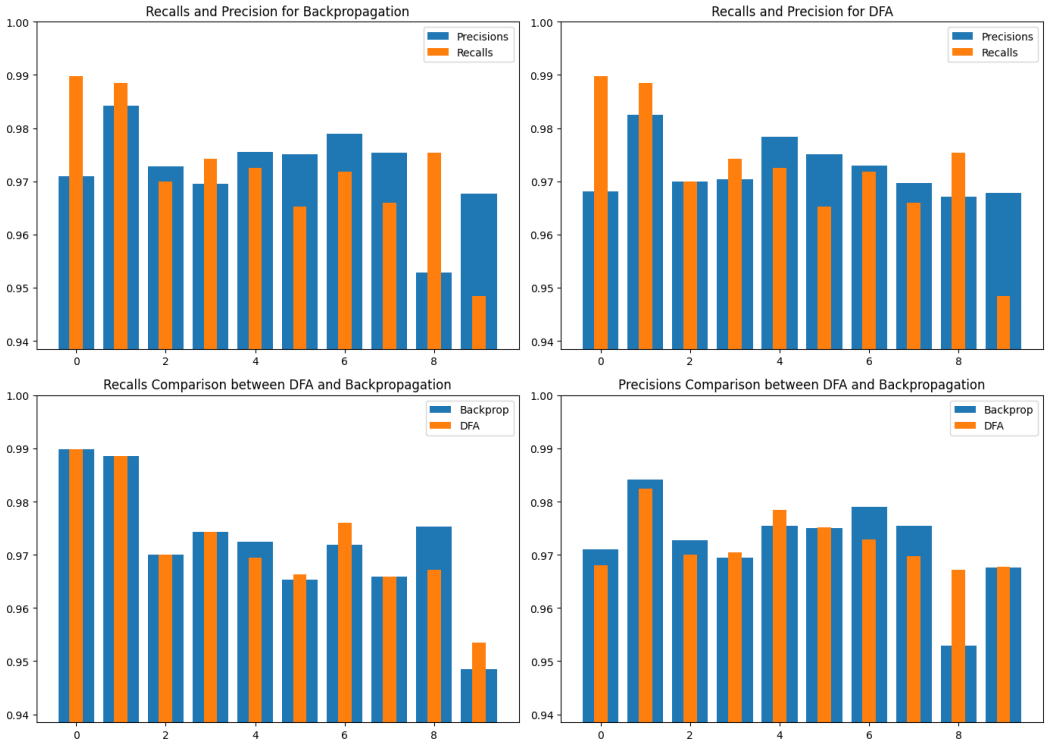

Figure 2: Precisions and recalls comparisons on an example run

We spot that for an example test case, which is a vanilla and unoptimized DFA and backprop, performance are the same with around a 97% accuracy on all digits. DFA looks more consistent though. We can conclude that this test validates our proof of concept, meaning that DFA works as expected by having equivalent performance to Backpropagation. Please note the difference in scales in Fig.1 which comes from the difference in how this value is estimated.

3 Beta Version: Framework

3.1 User guide

A Jupyter Notebook which goal is to introduce the worflow of the framework is provided in this project. The code follows a simple example trained on MNIST. Network model creation, including input, hidden layers and output sizes, as well as usage of the training loops, is covered. And boiler-plate testing code is also provided.

Please check the `framework_tutorial` for more informations.

3.2 Model Class

3.2.1 Initialization

The initialization of the `DynamicModel` class defines the structure and essential components of the model. The constructor takes the following inputs:

- `layer_sizes`: A list of integers specifying the number of neurons in each layer, including input, hidden, and output layers.

- `act_function`: The activation function to be applied between layers (e.g., `torch.nn.Tanh()`).
- `act_function_derivative`: The derivative of the activation function for gradient computations in the DFA backward pass.
- `output_function`: A function applied to the output layer (default is `torch.sigmoid`).

The code initializes a list of linear layers using `torch.nn.ModuleList`, which ensures that all layers are properly registered as part of the model. Additionally, the activation and output functions are stored as attributes for use in forward passes.

```
def __init__(self, layer_sizes, act_function, act_function_derivative, output_function=torch.
                                                    sigmoid):
    super(DynamicModel, self).__init__()
    self.layers = nn.ModuleList(
        [nn.Linear(layer_sizes[i], layer_sizes[i+1]) for i in range(len(layer_sizes) - 1)]
    )
    self.layer_sizes = layer_sizes
    self.act_function = act_function
    self.act_function_derivative = act_function_derivative
    self.output_function = output_function
```

Output: An instance of `DynamicModel`, ready to perform forward and backward passes.

3.2.2 Forward Passes

Two types of forward passes are implemented: one for simple use (without saving intermediate activations) and another for training, where intermediate activations are saved.

The training forward pass, `forward_pass_train`, propagates the input through the model, saving pre-activation values (a) and post-activation values (h) for every layer:

- `x`: The input tensor (h_0) to the network.
- `a`: A list of pre-activation values for each layer.
- `h`: A list of post-activation values for each layer.
- `y_hat`: The output predictions after applying the output function.

```
def forward_pass_train(self, x):
    a = []
    h = [x]
    for l_index in range(len(self.layers)-1):
        a.append(self.layers[l_index](h[-1]))
        h.append(self.act_function(a[-1]))

    a_last = self.layers[-1](h[-1])
    y_hat = self.output_function(a_last)
    return a, h, y_hat
```

The lighter forward pass is used when intermediate activations are not needed. Only the final predictions are returned:

- `x`: The input tensor.
- `y_hat`: The output predictions.

```
def forward(self, x):
    a = 0
    h = x
    for l_index in range(len(self.layers)-1):
        a = self.layers[l_index](h)
        h = self.act_function(a)

    a_last = self.layers[-1](h)
    y_hat = self.output_function(a_last)
    return y_hat
```

Output: In the training pass, intermediate values (a, h) and predictions (y_{hat}) are returned. In the simpler pass, only predictions (y_{hat}) are returned.

3.2.3 DFA Backward Pass

The backward pass computes weight and bias updates using Direct Feedback Alignment, relying on random feedback matrices (B). It calculates gradients for each layer using the error (e), intermediate activations (h), and pre-activations (a).

Inputs:

- `e`: Error tensor from the output layer.
- `h`: List of post-activation values for all layers, including the input.
- `a`: List of pre-activation values for all layers.
- `B`: A list of random feedback matrices for each layer.

Outputs:

- `dW`: A list of weight gradients for all layers.
- `db`: A list of bias gradients for all layers.

The computation involves the algorithm described in the scientific report, section 3.

```
def dfa_backward_pass(self, e, h, a, B):
    dW = []
    db = []

    for i in range(len(B)):
        da = torch.matmul(B[i], e) * self.act_function_derivative(a[i])
        dW.append(-torch.matmul(da, h[i].T))
        db.append(-torch.sum(da, dim=1, keepdim=True))

    dW.append(-torch.matmul(e, h[-1].T))
    db.append(-torch.sum(e, dim=1, keepdim=True))

    return dW, db
```

3.2.4 Printing a Summary of the Model

The `summary` method provides a detailed overview of the model's architecture, including layer indices, input and output shapes, and the activation functions used.

- `input_shape`: A tuple specifying the shape of the input tensor.

The method uses a dummy input tensor to trace the input and output shapes of each layer during a forward pass. For each layer, it prints the layer index, input shape, output shape, and activation type.

```
def summary(self, input_shape):
    print("============================MODEL SUMMARY==============================")
    print(f"{'Layer':<10}{'Input Shape':<20}{'Output Shape':<20}{'Activation':<20}")
    print("=" * 70)

    x = torch.rand(*input_shape)  # Dummy input to trace the shapes
    for i, layer in enumerate(self.layers):
        input_shape = x.shape
        x = layer(x)
        output_shape = x.shape
        activation = type(self.act_function) if i < len(self.layers) - 1 else type(self.
                                                        output_function)
        activation = str(activation)
        print(f"{i:<10}{str(list(input_shape)):<20}{str(list(output_shape)):<20}{activation:<20}"
                                                        )
    print("=" * 70)
    print(f'input size  : {self.layer_sizes[0]}')
    print(f'output size : {self.layer_sizes[-1]}')
    print(f'layers sizes: {self.layer_sizes}')
    print("============================END SUMMARY==============================")
```

Output: A printed table summarizing the structure of the model, including layer-wise details and input/output sizes.

3.3 Training loops

3.3.1 Basic training loop

The provided training loop implements Direct Feedback Alignment (DFA) in PyTorch for training a neural network. DFA replaces the standard backpropagation method by using random feedback matrices to compute gradients. Below is a detailed breakdown of each step in the code:

Inputs:

- `model`: A PyTorch model that implements `forward_pass_train` and `dfa_backward_pass`.
- `x, y`: Training data and labels. Both are converted to PyTorch tensors.
- `n_epochs`: Number of training epochs.
- `lr`: Learning rate for gradient updates.
- `batch_size`: Number of samples per batch.
- `tol`: Convergence tolerance based on changes in training error.
- `error_function`: Function to compute error and misclassification rate (e.g., `MNIST_error`).

Random Feedback Matrices Initialization :

```
B = [matrix_transfo(torch.empty(layer.out_features,
     model.layers[-1].out_features).normal_(mean=0,std=1.9)) for layer in model.layers[:-1]]
```

A list of random matrices (B) is generated for each layer except the output layer. The `matrix_transfo` function can optionally apply transformations to these matrices. We tried using them to skew the value distribution, without good results.

Dataset Preparation : The `TensorDataset` combines inputs (`x`) and targets (`y`) into a single dataset, while the `DataLoader` enables batch-wise data loading with shuffling to improve generalization.

Metrics Initialization :

```
te_dfa = []
loss_dfa = []
```

Two lists are initialized to record the training error (`te_dfa`) and loss (`loss_dfa`) for each epoch.

Epoch Loop: The outer loop iterates over the specified number of epochs (`n_epochs`). Within each epoch, the following steps are executed:

Batch Loop: The `DataLoader` provides batches of data (`batch_x` and `batch_y`), iterating through the entire dataset. This part can be linked to the scientific report, section 3.

- **Forward Pass:** The model's `forward_pass_train` method computes:
  - Pre-activation values (a) for each layer (list).
  - Post-activation values (h), including the input ($h_0 = x$) (list).
  - Predictions (y_{hat}) after applying the output function(list).
- **Error Computation:** The error signal is computed as:

$$\text{error} = y_{\text{hat}} - \text{batch\_y}$$

  The `error_function` also calculates the number of misclassified samples and updates the cumulative training error. Since MNIST is used throughout this project, we use the following function to compute the errorr and training error.:

```
def MNIST_error(y_hat, batch_y, train_error):
    error = y_hat - batch_y
    preds = torch.argmax(y_hat, dim=1)
    truth = torch.argmax(batch_y, dim=1)
    train_error += (preds != truth).sum().item()

    return error, train_error
```

- **Loss Calculation:** The binary cross-entropy loss between predictions (y_{hat}) and ground truth labels (batch_y) is computed:

$$\text{Loss}_{\text{batch}} = \texttt{F.binary\_cross\_entropy}(y_{\text{hat}}, \text{batch\_y})$$

  BCELoss is chosen here as it is standard for classificattion tasks.

- **Transpose Activations:** To ensure compatibility with feedback matrices, pre-activation (a) and post-activation (h) tensors are transposed (a^T and h^T).
- **DFA Backward Pass:** Using the `dfa_backward_pass` method, the weight gradients (dW) and bias gradients (db) are computed. Please check scientific report, section 3, for further informations.
- **Manual Weight Updates:** Model weights and biases are updated manually using the computed gradients and learning rate (η):

$$W_i \leftarrow W_i + \eta dW_i, \quad b_i \leftarrow b_i + \eta db_i$$

Epoch Metrics and Convergence Check: The total epoch loss (`epoch_loss`) and training error (`training_error`) are recorded. If the change in training error falls below the specified tolerance (`tol`), training terminates early.

```
if np.abs(training_error - prev_training_error) <= tol:
    print(f'Hitting tolerance of {tol} with {np.abs(training_error - prev_training_error)}')
    break
```

Output: The function returns two lists:

- `te_dfa`: Training error for each epoch.
- `loss_dfa`: Loss for each epoch.

The results can be used for further analysis or plotting.

3.3.2 Batch-averaged training loop

In this modified version of the training loop, the inputs, activations, pre-activations, and outputs are averaged across the batch to compute a "master error." This approach differs from the previous loop by aggregating batch-level information into a single representative value, which is then used for error computation and gradient updates. Below is a detailed explanation of the changes:

Relevant Code for Averaging: The following lines have been added to the batch loop:

```
for i in range(len(a)):
    a[i] = torch.mean(a[i], dim=0).unsqueeze(0)

for i in range(len(h)):
    h[i] = torch.mean(h[i], dim=0).unsqueeze(0)

y_hat = torch.mean(y_hat, dim=0).unsqueeze(0)
batch_y = torch.mean(batch_y, dim=0).unsqueeze(0)
batch_x = torch.mean(batch_x, dim=0).unsqueeze(0)

# Recompute error using averaged batches
error = y_hat - batch_y
```

Changes Compared to the Original Loop:

- **Averaging Pre-Activations (a) and Activations (h):** The pre-activations (a) and activations (h) are averaged across the batch along the first dimension (`dim=0`). Each resulting tensor is reshaped into a single-row tensor using `unsqueeze(0)`.
- **Averaging Outputs (y_{hat}) and Ground Truth (batch_y):** The predicted outputs (y_{hat}) and ground truth labels (batch_y) are also averaged along the batch dimension to produce a single representative prediction and target.
- **Averaging Inputs (batch_x):** Similarly, the batch inputs (batch_x) are averaged to obtain a single representative input vector for the entire batch.
- **Recomputing Error:** After averaging, the error is recomputed as:

$$\text{error} = y_{\text{hat}} - \text{batch\_y}$$

This averaged error serves as the "master error" for Direct Feedback Alignment.

3.3.3 Per class batch-averaged training loop

This modified batch loop introduces class-wise averaging for inputs, activations, pre-activations, and outputs within each batch. The primary goal is to compute class-level representations and update the model based on these averaged values, rather than individual sample-level gradients. Below, the differences from the previous loop are explained.

Class-Wise Indexing :

```
class_indices = [ [] for _ in range(classes)]
for item in range(len(batch_y)):
    class_indices[np.argmax(batch_y[item])].append(item)
```

This section initializes a list of indices for each class (`class_indices`). For every sample in the batch, the class corresponding to its encoded label (`batch_y`) is determined using `np.argmax`, and the sample's index is added to the appropriate class list.

Class-Wise Averaging :

```
for indices in class_indices:
    if indices == []:
        continue

    subset_a = []
    subset_h = []

    for i in range(len(a)):
        subset_a.append(torch.mean(a[i][indices], dim=0).unsqueeze(0))

    for i in range(len(h)):
        subset_h.append(torch.mean(h[i][indices], dim=0).unsqueeze(0))

    sub_y_hat = torch.mean(y_hat[indices], dim=0).unsqueeze(0)
    sub_batch_y = torch.mean(batch_y[indices], dim=0).unsqueeze(0)
```

This loop processes each class separately:

- If no samples belong to a class (`indices == []`), the loop skips to the next class.
- For each layer, pre-activations (a) and post-activations (h) of the samples in the class are averaged along the batch dimension (`dim=0`). The averaged tensors are unsqueezed to retain the correct shape for subsequent computations.
- Similarly, predictions (y_{hat}) and ground truth labels (batch_y) are averaged over the class's samples.

Recomputed Error Signal :

```
error = sub_y_hat - sub_batch_y
```

The error is recomputed based on the class-wise averaged predictions and labels. This replaces the sample-level error from the previous loop.

Key Differences from the Previous Loop:

- **Class-Averaging:** Unlike the previous loop, where computations were performed on individual samples, this loop averages inputs, activations, and outputs over all samples belonging to the same class within a batch.

- **Reduced Gradient Updates:** Gradient updates are performed at the class level, reducing the frequency of updates compared to the sample-wise approach.
- **Improved Stability:** By aggregating information at the class level, the updates are expected to be less noisy, potentially improving the training stability and convergence.
- **Error Signal Recalculation:** The error is recalculated using class-averaged predictions and labels, ensuring consistency with the aggregated representations.

Benefits of the New Approach: Class-wise averaging reduces the variability in gradient computations caused by individual samples. This can lead to more robust updates, especially in scenarios with imbalanced or noisy data. Additionally, this method aligns the training process with the objective of minimizing class-level errors rather than individual sample errors. The behavior of this approach ought to be studied separately from this project alone, as it could lead to interesting result.

4 How to use the provided demos

Notebooks Notebooks can be run as is, granted that dependencies are present and well configured on the host machine. In case of issues, please check the error messages as it most likely will be a missing dependency.

Python Scripts Python scripts are provided as is as an example. They should however work right out of the box, granted that dependencies are present and well configured on the host machine. While scripts work, we encourage the user to fiddle around the code. Some part have been coomented out and can leave some room for experimentation. We do assume the user to already be familiar with the Technical report content, especially regarding training loop functions' signatures.

5 Free Project versioning breakdown

5.0.1 Alpha

There are 2 notebooks that answers to the goals set by the Alpha version. The XOR test shows that DFA can be used to solve non-linear problem but do not provide much insight when comparing the DFA and Backpropagation. For this, please look at the MNIST test, which uses a more complex network for digits recognition in an image. Those 2 notebooks show a basic implementation of DFA, solve 2 simple problems, and compare performances with DFA. Thus it checks all requirements fro the Alpha. More advanced comparison between DFA and Backpropagation can be found in the scientific report.

5.0.2 Beta

Due to the nature of the project being focused on research, the goaals of this section shifted during time. While it was at first planned to analyze more in depth the difference in performance between DFA and Backpropagation, it was decided that it would be more fruitful to analyze a novel approach in training, using averages on batches. However, please note that while being succintly answered to, the requiremnts of this section have been completed. Scripts found in the Beta files do allow for the comparison of the angles (cf. Section 4.4 for analysis). The behavior difference between DFA and Backpropagation have been studied under the approach of averaging batches. Finally, a framework has been developped to ease the use of DFA in project. This framework was developped alongside the test files.

5.0.3 Gold

Gold requirements weren't met in any capacity.