Greedy Subspace Localization

(TODO: Better Title)

Akhil Sadam[1], **Dr.Tan Bui Thanh**[2]

[1]Department of Aerospace Engineering and Engineering Mechanics, UT Austin
[2]The Oden Institute for Computational Sciences, UT Austin

E-mail: akhil.sadam@utexas.edu, tanbui@oden.utexas.edu
April 2023

1. Motivation

1.1. Background

Contrastive learning procedures commonly use out-of-distribution (OOD) samples to train classifiers, as in Dr. Hinton's recent work [**?**]. OOD samples are also used to train regressive models. An example would be Monte-Carlo null-space estimation for large linear systems (cite, better example?).

No regressive OOD, or negative data approaches are done in a greedy manner, and they do not separate regression from localization (is this true ?). We define localization as the process of subspace generation and feature selection, and regression as the process of fitting a model to the selected features.

The following goals are met by our approach:

- Greedy Layerwise Learning
- Separation of Localization and Regression
- OOD-based Data Augmentation for Localization

(not sure if I should go through the FFA and motivate this work, or go directly to the architecture.)

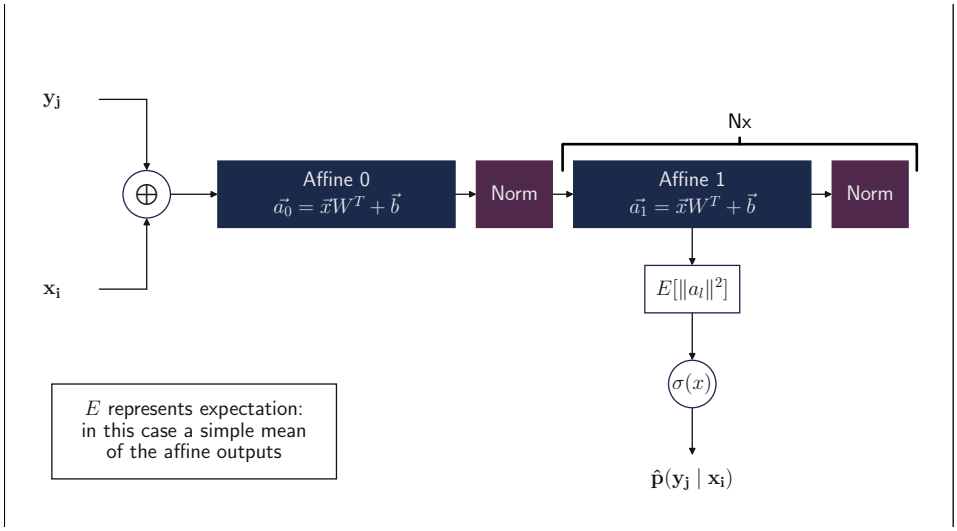
1.2. FFA-styled Contrastive Learning

Recall Dr. Hinton's recent work [**?**]:

Assume a supervised problem with m labels and t test samples as below.

$$\mathbf{X_{test}} = \{x_i \mid i \in 1, ..., t\},\ \mathbf{Y_{labels}} = \{y_j \mid j \in 1, ..., m\} \tag{1.1}$$

In prediction mode, each test sample x_i is combined with every label $y_j, \forall j \in 1, ..., m$. The net [Fig. 1a] outputs m predictions of $\hat{p}(y_j \mid x_i)$. For a single-label classifier, the label with the highest probability is chosen. To classify all samples the FFA requires $m * t$ runs. Normalization is required after each layer to ensure that



(a) Supervised FFA Architecture

prior layer $\|a_{l-1}\|^2$ does not affect later layers. Training is a greedy layer-by-layer approach.

Clearly, regression is not possible with this architecture. Several adjustments will now be made to allow for regression. Since y_j cannot be input unless each layer is to calculate an integral, we remove the direct product, and learn y_j directly, removing the final sigmoid activation. Note the overall function of the Affine + Norm layer is to shape / localize the input space, and layer norms no longer work as a layerwise loss. We replace this with a localization layer, which is a standard Affine layer with a Lipschitz loss function. Finally, we add regression layers, to replace the expectation, resulting in a contrastive layerwise-regression architecture, that can incorporate OOD sampling.

2. Base Forward Decomposition (FD) for supervised regression

2.1. Architecture

Assume a supervised problem where the y-function is to be learnt.

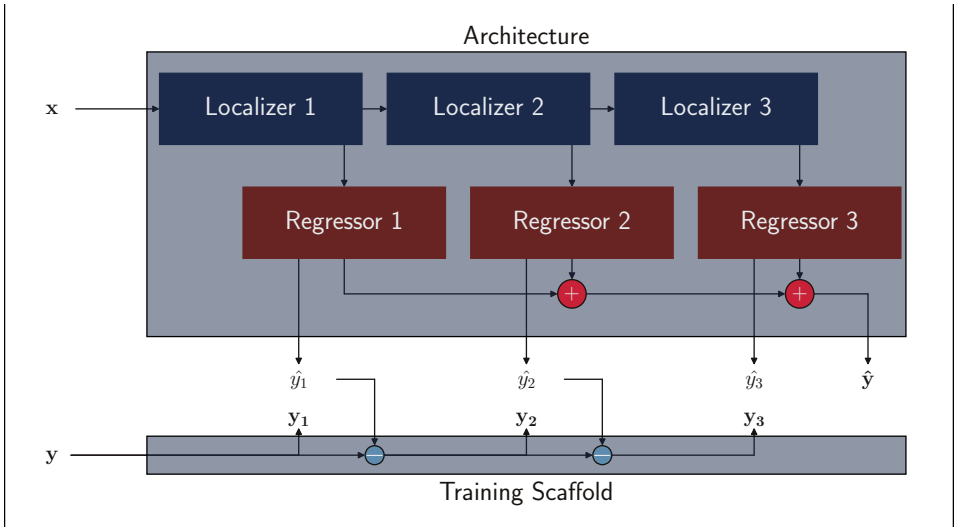
$$\mathbf{D} = \{(x_i, y_i) \mid y_i = \mathbf{F_y}(x_i) \, \forall \, i \in 1, ..., n\} \tag{2.1}$$

A general 3-layer implementation is shown in Figure 1a.

2.2. Loss Functions and Training

2.2.1. Base Localizer Loss Given input, output vector spaces X and Y, with an intermediate, localized space Z, we define the localizer and regressor as L, R: Note F_y denotes the mapping to be learnt (as before).

$$L : X \to Z, R : Z \to Y, F_y : X \to Z. \tag{2.2}$$



(a) Base FDA Architecture

For noise-augmented x with a variance of δ,

$$\epsilon_p \sim N(0, \delta^2) \tag{2.3}$$

$$\hat{y_{aug}} = R(z), z = L(x + \epsilon_p) \tag{2.4}$$

To localize the input space X into Z, given an approximate Lipschitz constant, and a particular neighborhood size (a delta-ball), the following loss is used. Note σ denotes the ReLU function, and MSE denotes mean-squared-error. Both the prediction and the true y-value are lifted to a higher space by concatenation (denoted by $\oplus$) with the latent-space value z, so that the continuity condition imposed can work with discontinous functions.

$$\text{Loss}_{\text{localizer}}(\mathbf{x}, \mathbf{y}) = \sigma(\text{MSE}(\hat{y_{aug}} \oplus z, y \oplus z) - k^2\delta^2) \tag{2.5}$$

Note this condition follows directly from the Lipschitz continuity - that the output from the δ-ball falls within the $k\delta$-ball. The output ball is centered at the true y-value instead of the mean, due to computational ease; this also serves as a MSE addition.

2.2.2. OOD Data Augmentation (TODO remove colloquial language) To allow for contrastive learning, and accurately learn the 'nullspace', or space not data-driven, negative-data is generated as follows:

$$x_n = x + \epsilon_n \tag{2.6}$$

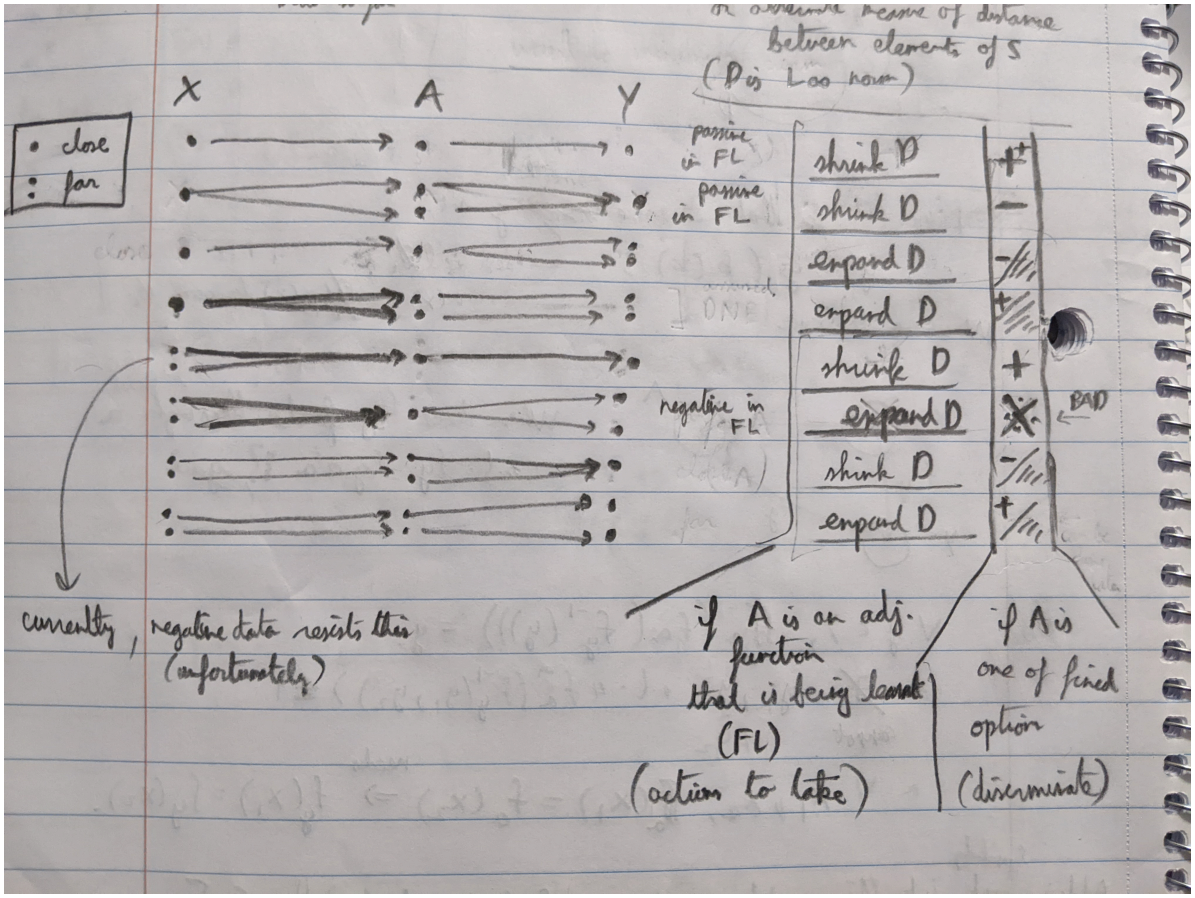
$$\|\epsilon_n\| = \delta + \epsilon_{\text{machine-precision}}, \text{in random direction} \tag{2.7}$$

This negative data allows an additional localizer loss component:

$$\text{Loss}_{\text{localizer-negative}}(\mathbf{x_n}, \mathbf{y}) = \sigma(-\text{MSE}(\hat{y_{aug}} \oplus z, y \oplus z) + k^2\delta^2) \tag{2.8}$$

2.2.3. Variance Expansion Loss (TODO) Greatly limited function behavior has been accounted for. For example, a maximum-variance expansion like PCA would be useful in some cases. If neighbors in Y-space are considered, their set diameter in Z-space denoted D, the following results for some fixed $y \in Y$:

$$\mathbf{D} = \text{diam}(L(f_y^{-1}(N_\varepsilon(y)))) \tag{2.9}$$



(TODO need another picture for X,Y,Z relationship with delta,epsilon-balls) (TODO replace photo, change A to Z) (TODO add loss for variance-expansion) (TODO rewrite with more text)

2.2.4. Regressor Loss On the regression side, a straightforward MSE loss is implemented.

$$\text{Loss}_{\text{regressor}}(\mathbf{x}, \mathbf{y}) = \text{MSE}(\hat{y_{aug}}, y) \tag{2.10}$$

2.2.5. Layerwise Training Notice the above deals with a single layer. Several layers can be stacked and trained in a greedy fashion, by freezing prior weights and learning the residual.

2.2.6. Complete Algorithm (TODO rewrite above in a algorithm form)

2.3. Theorems?

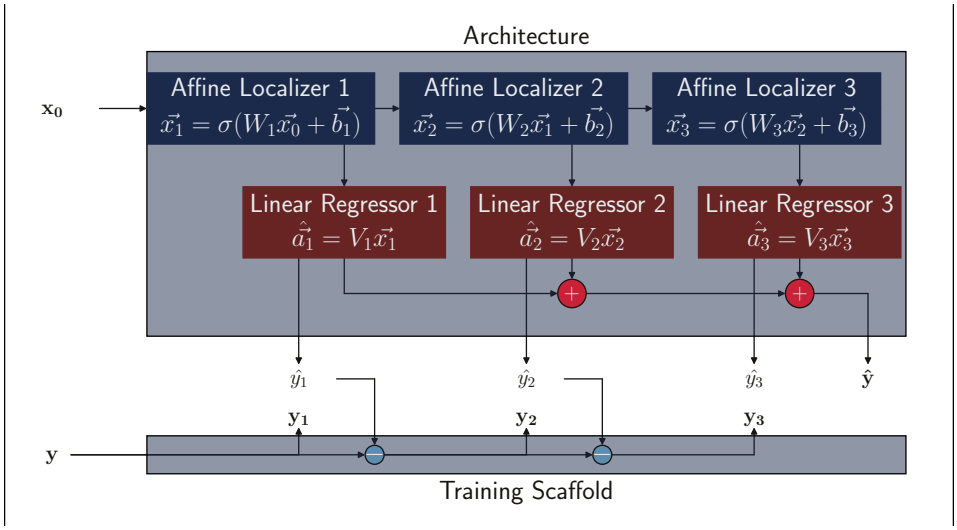
Note this method is similar to ResNet, so can we avoid gradient issues? Any convergence theorems?

2.4. UQ?

Uncertainty quantification?

2.5. Results

We use standard ReLU layers for the localizer and linear layers for the regressor. 1a gives the following.



(a) Base FDA Architecture

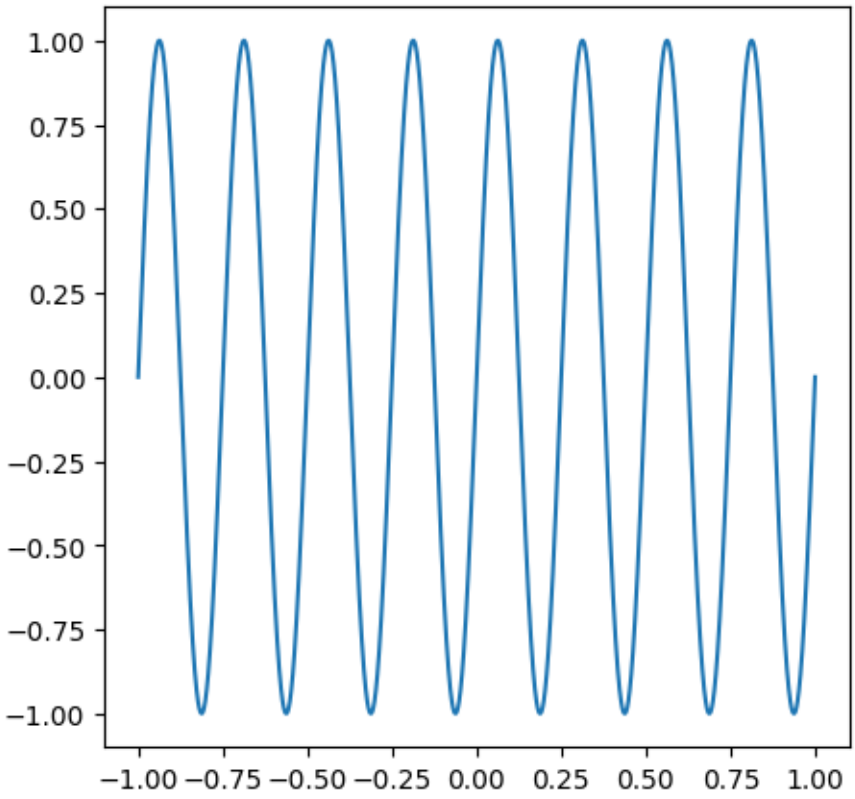
(TODO - rewrite section with curated examples once the algorithm is finalized)

2.5.1. Convergence Rates at different frequencies with noisy data

hf_lf_v2

April 16, 2023

[ ]:
```
import numpy as np
from numba import jit
from numpy.random import default_rng as rg
rng = rg(12345)
from tqdm import tqdm
import sys
sys.path.append('../core')
%matplotlib inline
import torch
torch.manual_seed(0)
import torch.optim as optim
from FLDojo import dojo
from FL import FL
from DNN_R import DNN
from display2 import*
Xs = np.linspace(-1,1,40000)
Ys = np.sin(Xs*8*np.pi)
from matplotlib import pyplot as plt
fig,axs = plt.subplots(1,1,figsize=(5,5))
axs.plot(Xs,Ys)

dnn_sizes = [1,102,101,1] # so # of weights is 102*1 + 102*102 + 102*1 = 10608
fl_sizes = [1,100,100] # so # of weights is  1*a + a*1 + a*a + a*1 = 10300, for␣
 ↪a = 100
fl_adapt_sizes = lambda k : [1,k*32,k*32] # so # of weights is  1*a + a*1 + a*a␣
 ↪+ a*1 = 10300, for a = 100

fl_sizes2 = [1,25,25] # so # of weights is  1*a + a*1 + a*a + a*1 = 700, for a␣
 ↪= 25
dnn_sizes2 = [1,26,26,1] # so # of weights is 26*1 + 26*26 + 26*1 = 728
# a three layer nn can represent any multivariate function (continuous or␣
 ↪discontinuous) https://arxiv.org/abs/2012.03016
```



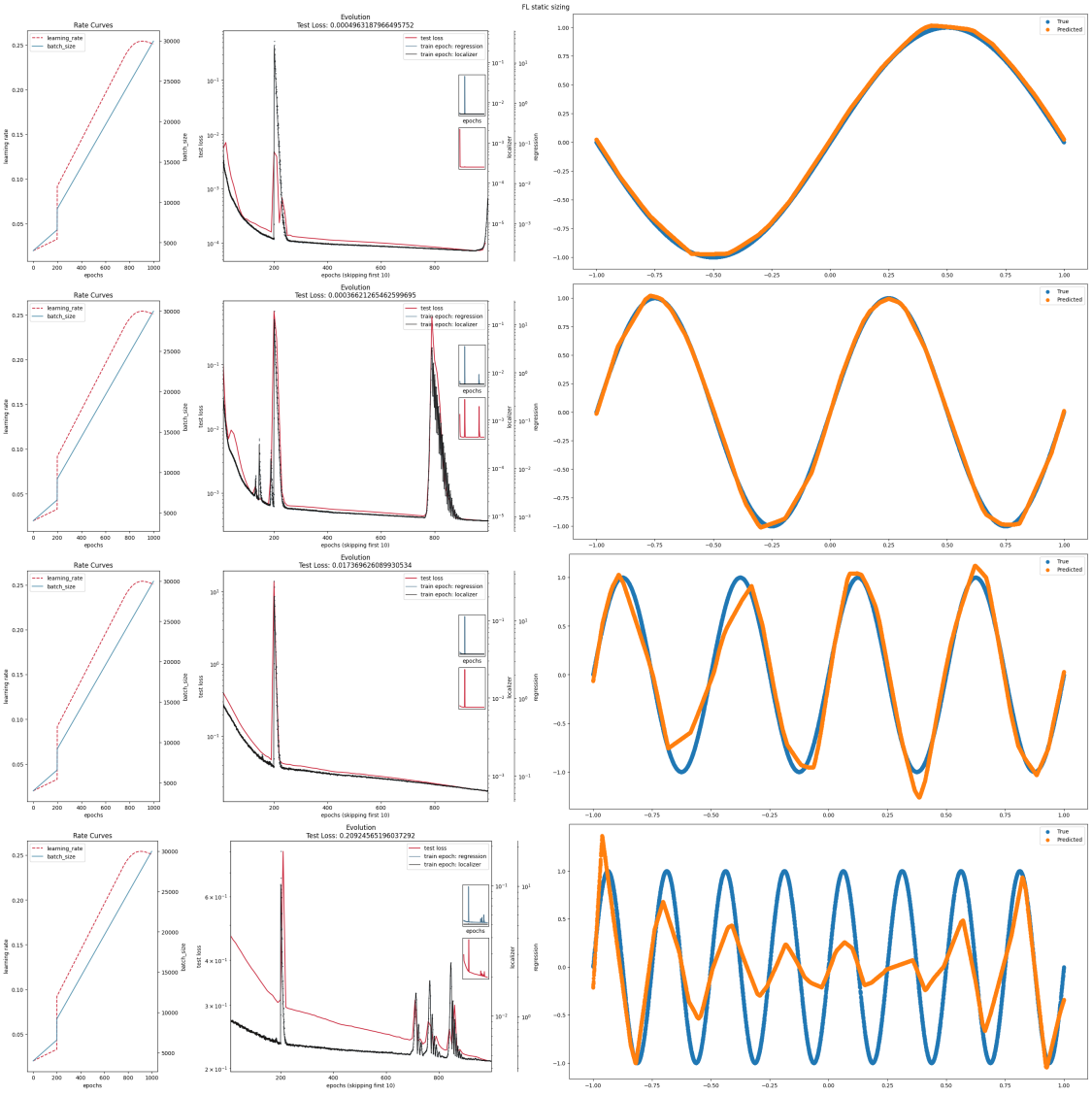
[ ]:
```
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
p = np.random.permutation(len(Xs))
Xs = Xs[p]
X = torch.from_numpy(Xs).float().unsqueeze(1).to(device)
split = 0.75
train_X = X[:int(split*len(X))]
test_X = X[int(split*len(X)):]

def update_y(k):
    Ys = np.sin(Xs*k)
    y = torch.from_numpy(Ys).float().unsqueeze(1).to(device)
    train_y = y[:int(split*len(y))]
    test_y = y[int(split*len(y)):]
    return train_y, test_y
```

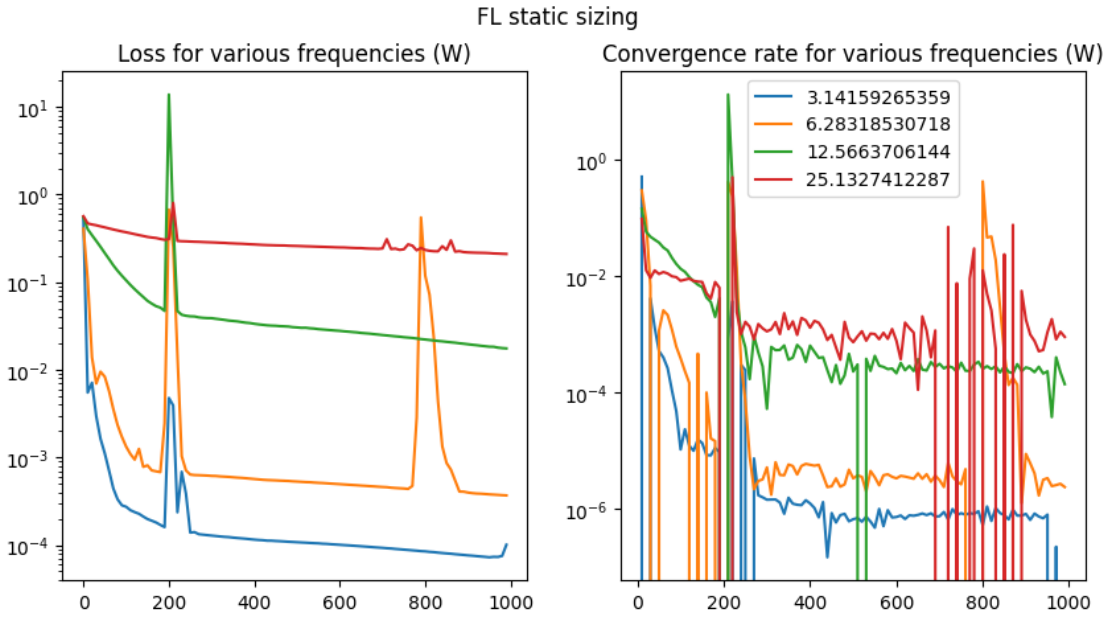
[ ]:
```
D = dojo()
D.epochs=1000
D.max_batch_size=train_X.shape[0]
opt = lambda x: optim.Adam(x, lr=0.00001) # Adam better than SGD and AdamW in␣
 ↪quick tests.
act = torch.nn.ReLU()
```

```
delta = np.array([1]*len(fl_sizes))*0.00005
```

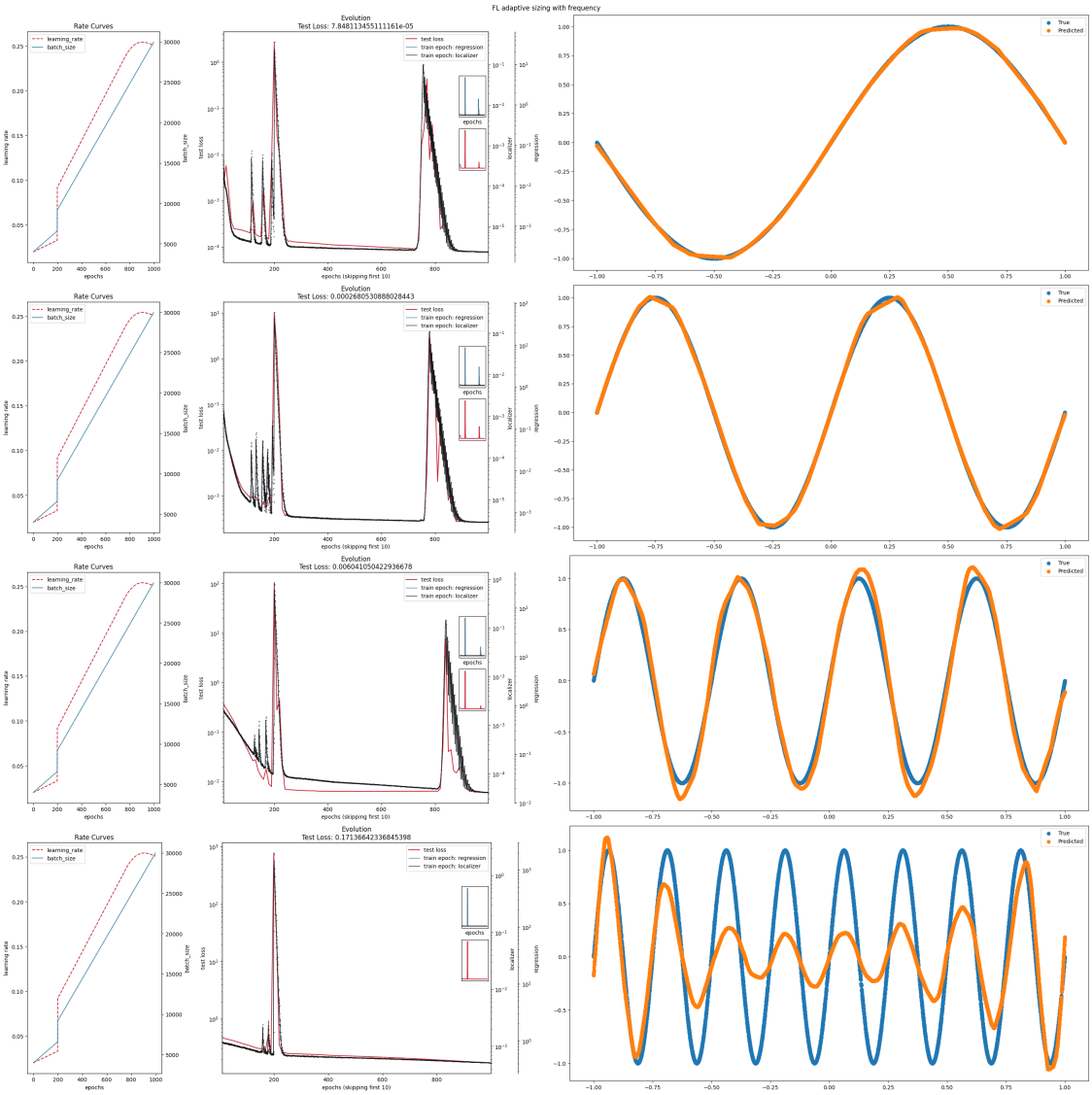
[ ]:
```
reports = []
ks_ = np.array([1,2,4,8])*np.pi
for j in range(2):
    st = 'FL static sizing' if j == 0 else 'FL adaptive sizing with frequency'
    fig = plt.figure(layout='constrained', figsize=(28,28))
    subfigs = fig.subfigures(len(ks_),1, hspace=0.01)
    for i,ks in enumerate(ks_):
        train_y,test_y = update_y(ks)


        k = [ks]*len(fl_sizes)
        if j == 0:
            net = FL(device, fl_sizes, delta, k, opttype=opt, act = act,␣
 ↪bias=True)
        else:
            net = FL(device, fl_adapt_sizes(int(ks)), delta, k, opttype=opt,␣
 ↪act = act, bias=True)
        report = D.train(net, train_X, train_y, test_X, test_y,␣
 ↪start_batch_size=4000)
        sfgs = subfigs[i].subfigures(1,2)
        ecran(net, test_X, test_y, report, classification=False,subfig =␣
 ↪sfgs[0])
        p1 = sfgs[1].subplots(1,1)
        p1.scatter(test_X.detach().cpu().numpy(),test_y.detach().cpu().
 ↪numpy(),label='True')
        p1.scatter(test_X.detach().cpu().numpy(),net(test_X).detach().cpu().
 ↪numpy(),label='Predicted')
        p1.legend()
        reports.append(report)
    fig.suptitle(st)
    plt.show()
    plt.close(fig)
    fig,axs = plt.subplots(1,2,figsize=(10,5))
    axs[0].set_title('Loss for various frequencies (W)')
    axs[1].set_title('Convergence rate for various frequencies (W)')
    for i in range(len(ks_)):
        axs[0].plot(reports[i][-2],reports[i][-1],label=ks_[i])
        axs[1].plot(reports[i][-2][1:],-np.diff(reports[i][-1]),label=ks_[i])
        axs[0].set_yscale('log')
        axs[1].set_yscale('log')
    fig.suptitle(st)
    plt.legend()
    plt.show()
    plt.close(fig)
```

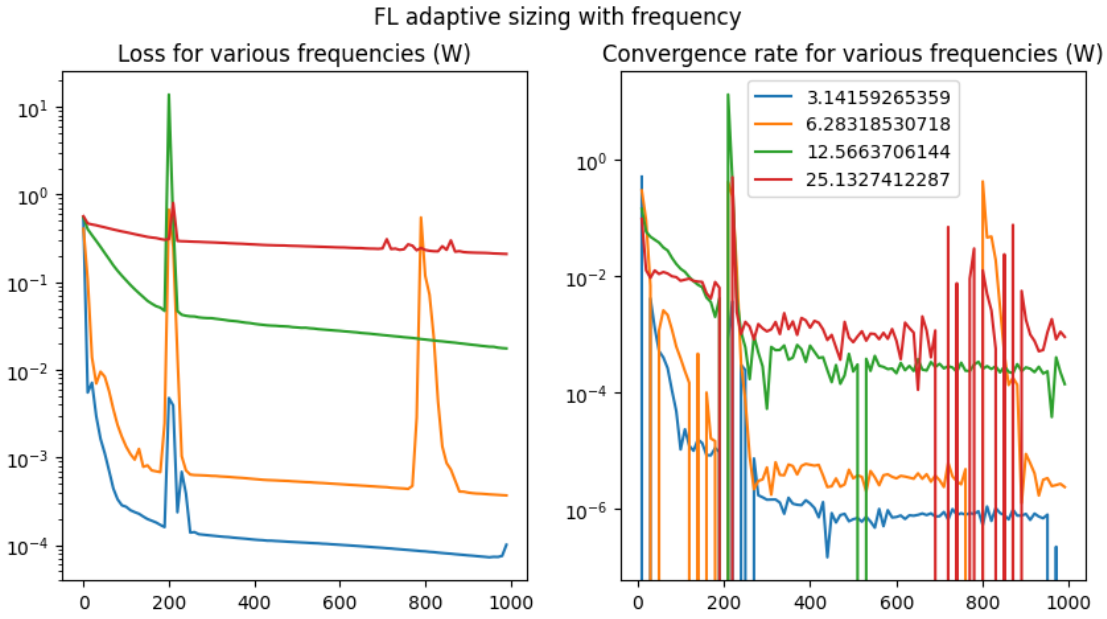
```
100%|      | 1000/1000 [00:34<00:00, 29.24it/s]
100%|      | 1000/1000 [00:33<00:00, 29.94it/s]
100%|      | 1000/1000 [00:32<00:00, 30.49it/s]
100%|      | 1000/1000 [00:34<00:00, 29.19it/s]
```






```
100%|      | 1000/1000 [00:31<00:00, 31.93it/s]
100%|      | 1000/1000 [00:54<00:00, 18.44it/s]
100%|      | 1000/1000 [01:49<00:00,  9.09it/s]
100%|      | 1000/1000 [05:02<00:00,  3.31it/s]
```


FL adaptive sizing with frequency
Rate Curves
learning_rate
batch_size
epochs
Evolution
Test Loss: 7.848113455111161e-05
test loss
train epoch: regression
train epoch: localizer
epochs (skipping first 10)
True
Predicted
Rate Curves
learning_rate
batch_size
epochs
Evolution
Test Loss: 0.0002680530888028443
test loss
train epoch: regression
train epoch: localizer
epochs (skipping first 10)
True
Predicted
Rate Curves
learning_rate
batch_size
epochs
Evolution
Test Loss: 0.006041050422936678
test loss
train epoch: regression
train epoch: localizer
epochs (skipping first 10)
True
Predicted
Rate Curves
learning_rate
batch_size
epochs
Evolution
Test Loss: 0.17136642336845398
test loss
train epoch: regression
train epoch: localizer
epochs (skipping first 10)
True
Predicted

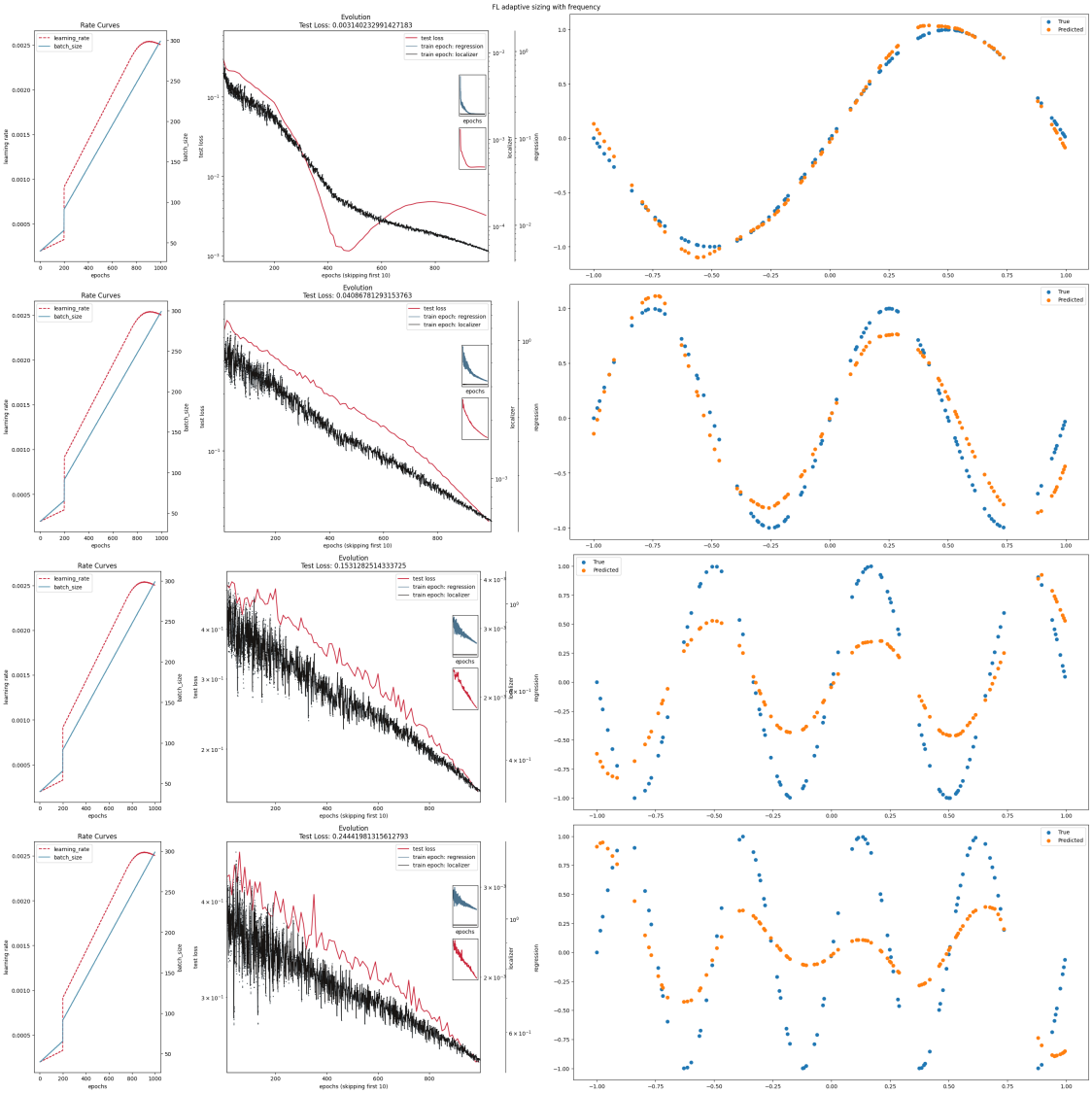


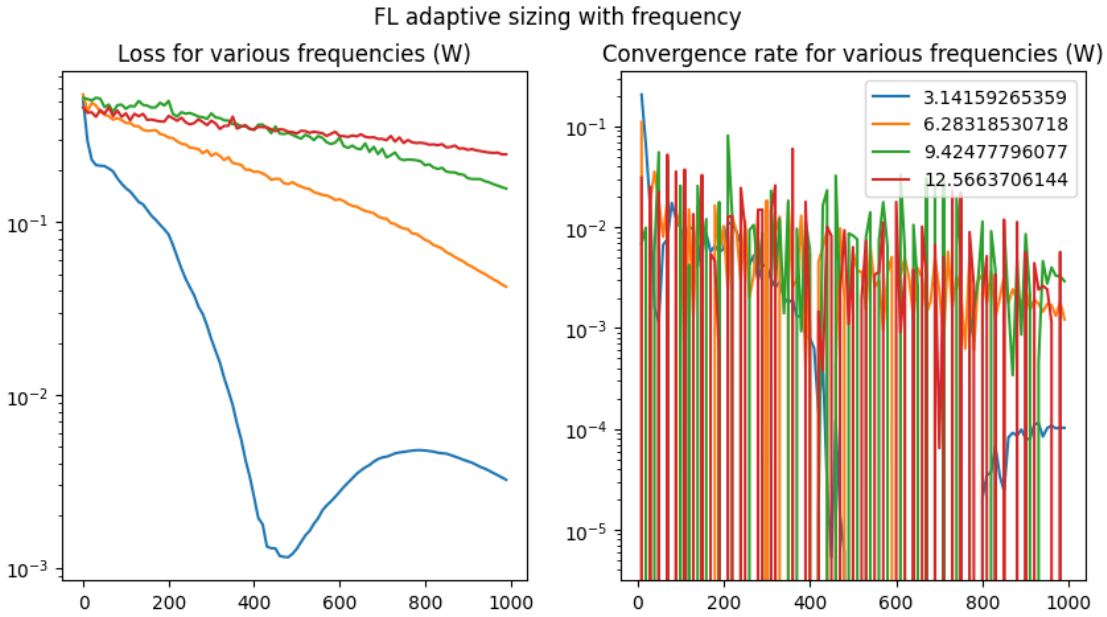
[ ]:
```
D.epochs=1000
D.max_batch_size=train_X.shape[0]
opt = lambda x: optim.Adam(x, lr=0.00001) # Adam better than SGD and AdamW in␣
 ↪quick tests.
```

[ ]:
```
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
Xs = np.linspace(-1,1,400)
p = np.random.permutation(len(Xs))
Xs = Xs[p]
X = torch.from_numpy(Xs).float().unsqueeze(1).to(device)
split = 0.75
train_X = X[:int(split*len(X))]
test_X = X[int(split*len(X)):]
def update_y2(k):
    Ys = np.sin(Xs*k)
    y = torch.from_numpy(Ys).float().unsqueeze(1).to(device)
    train_y = y[:int(split*len(y))]
    test_y = y[int(split*len(y)):]
    return train_y, test_y

reports = []
ks_ = np.array([1,2,3,4])*np.pi

for j in range(1,2):
    st = 'FL static sizing' if j == 0 else 'FL adaptive sizing with frequency'
    fig = plt.figure(layout='constrained', figsize=(28,28))
```

```
    subfigs = fig.subfigures(len(ks_),1, hspace=0.01)
    for i,ks in enumerate(ks_):
        train_y,test_y = update_y2(ks)


        k = [ks]*len(fl_sizes)
        if j == 0:
            net = FL(device, fl_sizes, delta, k, opttype=opt, act = act,␣
↪bias=True)
        else:
            net = FL(device, fl_adapt_sizes(int(ks)), delta, k, opttype=opt,␣
↪act = act, bias=True)
        report = D.train(net, train_X, train_y, test_X, test_y,␣
↪start_batch_size=40)
        sfgs = subfigs[i].subfigures(1,2)
        ecran(net, test_X, test_y, report, classification=False,subfig =␣
↪sfgs[0])
        p1 = sfgs[1].subplots(1,1)
        p1.scatter(test_X.detach().cpu().numpy(),test_y.detach().cpu().
↪numpy(),label='True')
        p1.scatter(test_X.detach().cpu().numpy(),net(test_X).detach().cpu().
↪numpy(),label='Predicted')
        p1.legend()
        reports.append(report)
    fig.suptitle(st)
    plt.show()
    plt.close(fig)
    fig,axs = plt.subplots(1,2,figsize=(10,5))
    axs[0].set_title('Loss for various frequencies (W)')
    axs[1].set_title('Convergence rate for various frequencies (W)')
    for i in range(len(ks_)):
        axs[0].plot(reports[i][-2],reports[i][-1],label=ks_[i])
        axs[1].plot(reports[i][-2][1:],-np.diff(reports[i][-1]),label=ks_[i])
        axs[0].set_yscale('log')
        axs[1].set_yscale('log')
    fig.suptitle(st)
    plt.legend()
    plt.show()
    plt.close(fig)
```

```
100%|      | 1000/1000 [00:14<00:00, 69.50it/s]
100%|      | 1000/1000 [00:13<00:00, 71.51it/s]
100%|      | 1000/1000 [00:14<00:00, 68.86it/s]
100%|      | 1000/1000 [00:15<00:00, 63.82it/s]
```

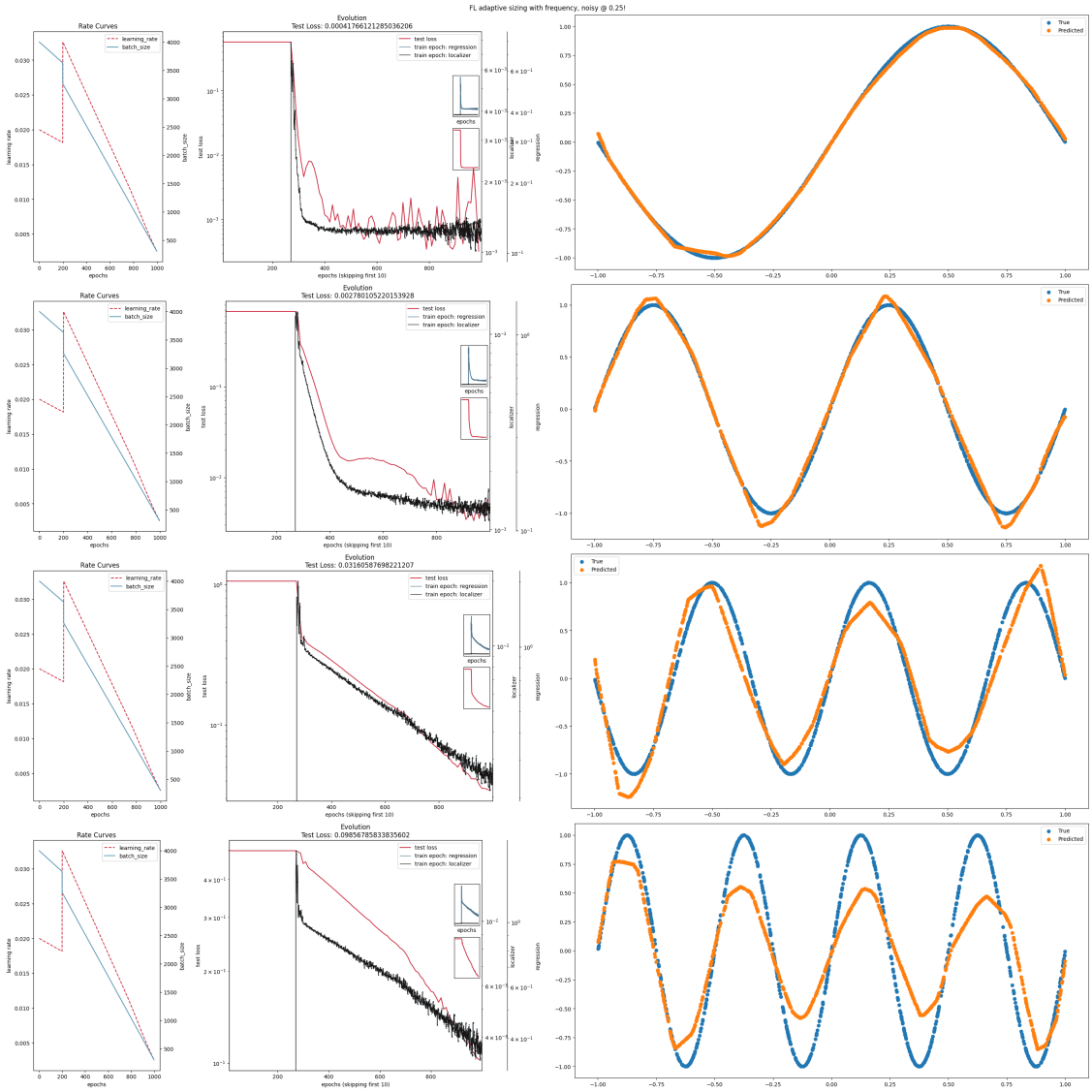

FL adaptive sizing with frequency
Rate Curves
learning_rate
batch_size
epochs
Evolution
Test Loss: 0.003140232991427183
test loss
train epoch: regression
train epoch: localizer
epochs (skipping first 10)
True
Predicted
Rate Curves
learning_rate
batch_size
epochs
Evolution
Test Loss: 0.0408678129315376 3
test loss
train epoch: regression
train epoch: localizer
epochs (skipping first 10)
True
Predicted
Rate Curves
learning_rate
batch_size
epochs
Evolution
Test Loss: 0.153128251433372 5
test loss
train epoch: regression
train epoch: localizer
epochs (skipping first 10)
True
Predicted
Rate Curves
learning_rate
batch_size
epochs
Evolution
Test Loss: 0.244419813156127 93
test loss
train epoch: regression
train epoch: localizer
epochs (skipping first 10)
True
Predicted

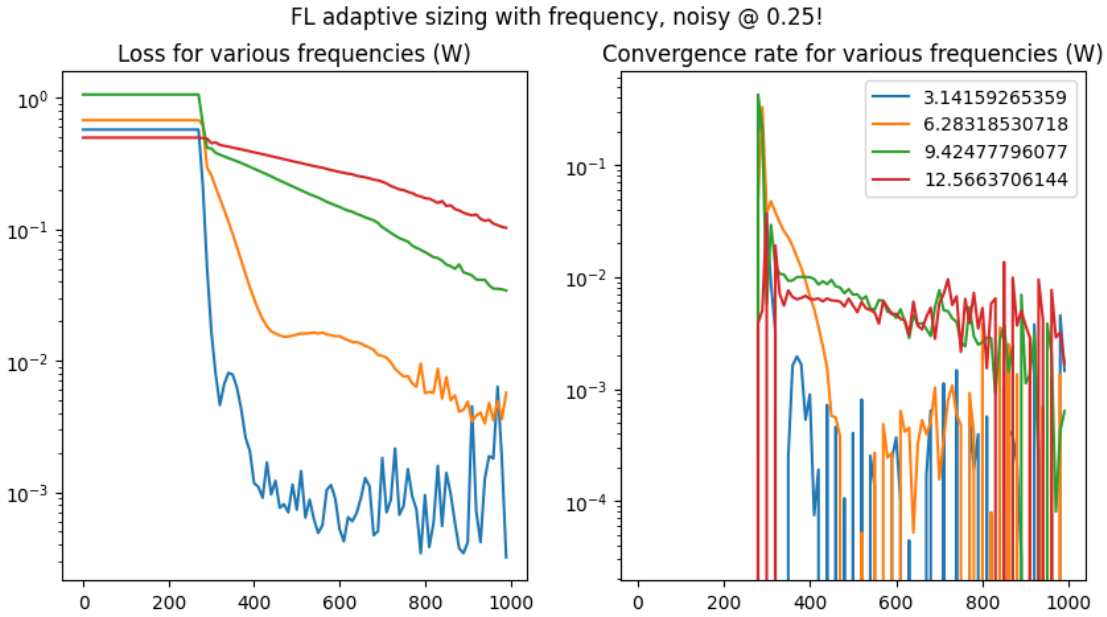


[ ]:
```
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
Xs = np.linspace(-1,1,4000)
p = np.random.permutation(len(Xs))
Xs = Xs[p]
X = torch.from_numpy(Xs).float().unsqueeze(1).to(device)
split = 0.75
train_X = X[:int(split*len(X))]
test_X = X[int(split*len(X)):]
def update_y3(k, noise):
    Ys = np.sin(Xs*k)
    y = torch.from_numpy(Ys).float().unsqueeze(1).to(device)
    train_y = y[:int(split*len(y))] + torch.randn_like(y[:
 ↪int(split*len(y))])*noise
    test_y = y[int(split*len(y)):]
    return train_y, test_y

reports = []
ks_ = np.array([1,2,3,4])*np.pi
noises = [0.25,0.5]
for j in range(0,2):
    noise = noises[j]
    st = f'FL adaptive sizing with frequency, noisy @ {noise}!'
    fig = plt.figure(layout='constrained', figsize=(28,28))
    subfigs = fig.subfigures(len(ks_),1, hspace=0.01)
    for i,ks in enumerate(ks_):
        train_y,test_y = update_y3(ks, noise)
```

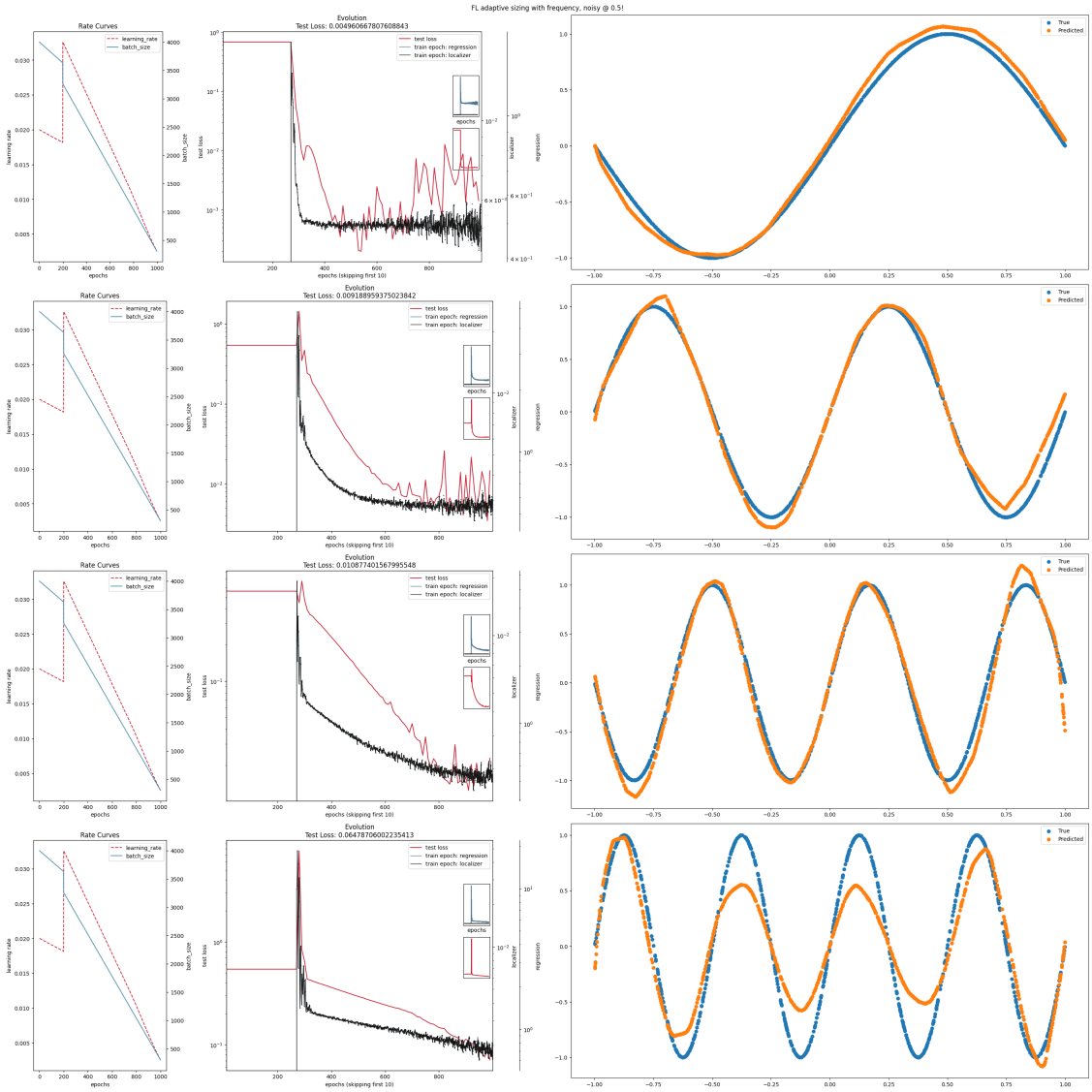
```
        k = [ks]*len(fl_sizes)
        if j == 0:
            net = FL(device, fl_sizes, delta, k, opttype=opt, act = act,␣
↪bias=True)
        else:
            net = FL(device, fl_adapt_sizes(int(ks)), delta, k, opttype=opt,␣
↪act = act, bias=True)
        report = D.train(net, train_X, train_y, test_X, test_y,␣
↪start_batch_size=4000)
        sfgs = subfigs[i].subfigures(1,2)
        ecran(net, test_X, test_y, report, classification=False,subfig =␣
↪sfgs[0])
        p1 = sfgs[1].subplots(1,1)
        p1.scatter(test_X.detach().cpu().numpy(),test_y.detach().cpu().
↪numpy(),label='True')
        p1.scatter(test_X.detach().cpu().numpy(),net(test_X).detach().cpu().
↪numpy(),label='Predicted')
        p1.legend()
        reports.append(report)
    fig.suptitle(st)
    plt.show()
    plt.close(fig)
    fig,axs = plt.subplots(1,2,figsize=(10,5))
    axs[0].set_title('Loss for various frequencies (W)')
    axs[1].set_title('Convergence rate for various frequencies (W)')
    for i in range(len(ks_)):
        axs[0].plot(reports[i][-2],reports[i][-1],label=ks_[i])
        axs[1].plot(reports[i][-2][1:],-np.diff(reports[i][-1]),label=ks_[i])
        axs[0].set_yscale('log')
        axs[1].set_yscale('log')
    fig.suptitle(st)
    plt.legend()
    plt.show()
    plt.close(fig)
```

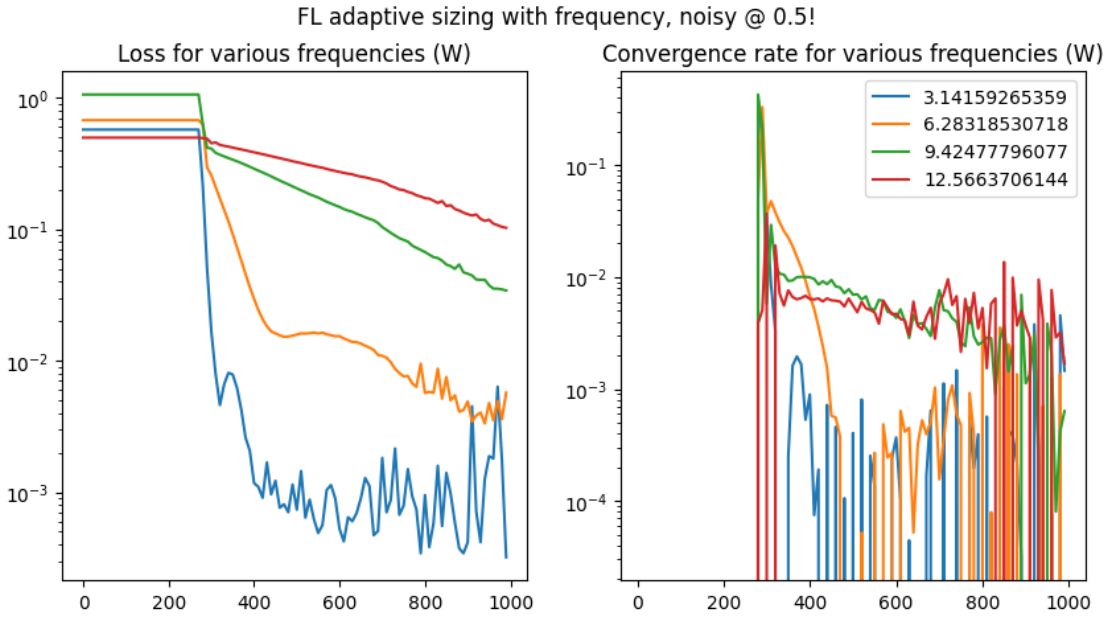
```
100%|      | 1000/1000 [00:11<00:00, 86.27it/s]
100%|      | 1000/1000 [00:10<00:00, 93.03it/s]
100%|      | 1000/1000 [00:11<00:00, 88.70it/s]
100%|      | 1000/1000 [00:10<00:00, 93.51it/s]
```


FL adaptive sizing with frequency, noisy @ 0.25!
Rate Curves
learning_rate
batch_size
epochs
Evolution
Test Loss: 0.00041766121285036206
test loss
train epoch: regression
train epoch: localizer
epochs (skipping first 10)
True
Predicted
Rate Curves
Evolution
Test Loss: 0.002780105220153928
Rate Curves
Evolution
Test Loss: 0.03160587698221207
Rate Curves
Evolution
Test Loss: 0.09856785833835602




```
100%|      | 1000/1000 [00:12<00:00, 81.53it/s]
100%|      | 1000/1000 [00:11<00:00, 85.75it/s]
100%|      | 1000/1000 [00:12<00:00, 77.99it/s]
100%|      | 1000/1000 [00:14<00:00, 69.54it/s]
```


FL adaptive sizing with frequency, noisy @ 0.5!
Rate Curves
learning_rate
batch_size
epochs
Evolution
Test Loss: 0.004960667807608843
test loss
train epoch: regression
train epoch: localizer
epochs (skipping first 10)
True
Predicted
Rate Curves
learning_rate
batch_size
epochs
Evolution
Test Loss: 0.009188959375023842
test loss
train epoch: regression
train epoch: localizer
epochs (skipping first 10)
True
Predicted
Rate Curves
learning_rate
batch_size
epochs
Evolution
Test Loss: 0.010877401567995548
test loss
train epoch: regression
train epoch: localizer
epochs (skipping first 10)
True
Predicted
Rate Curves
learning_rate
batch_size
epochs
Evolution
Test Loss: 0.06478706002235413
test loss
train epoch: regression
train epoch: localizer
epochs (skipping first 10)
True
Predicted


FL adaptive sizing with frequency, noisy @ 0.5!
Loss for various frequencies (W)
Convergence rate for various frequencies (W)
3.14159265359
6.28318530718
9.42477796077
12.5663706144

[ ]:

2.5.2. Comparison with DNN

compare_v2

April 16, 2023

[ ]:
```
import numpy as np
from numba import jit
from numpy.random import default_rng as rg
rng = rg(12345)
from tqdm import tqdm
import sys
sys.path.append('../core')
%matplotlib inline
```

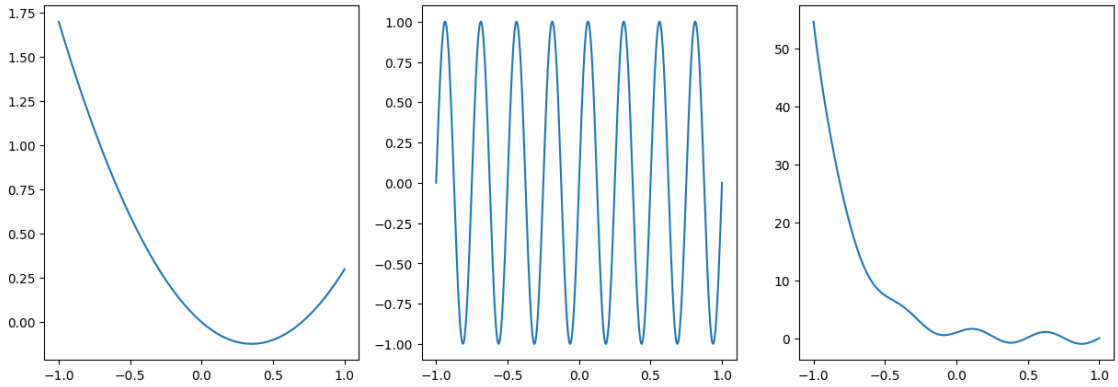
[ ]:
```
import torch
torch.manual_seed(0)
import torch.optim as optim
```

[ ]:
```
from FLDojo import dojo
from FL import FL
from DNN_R import DNN
from display2 import*
```

[ ]:
```
Xs = np.linspace(-1,1,40000)
Ys0 = Xs**2 - 0.7*Xs
# derivative is 2x - 0.7, which has a absval max of 2.7 (which is the Lipshitz␣
 ↪constant)
Ys1 = np.sin(Xs*8*np.pi)
# note derivative is 8 pi cos(8 pi x), which has a max of 8 pi (which is the␣
 ↪Lipshitz constant)
Ys2 = np.sin(Xs*4*np.pi) + np.exp(-4*Xs)
print(np.max(np.abs(4*np.pi*np.cos(4*np.pi*Xs) - 4*np.exp(-4*Xs))))
# derivative has a max of ~205 (which is the Lipshitz constant)
from matplotlib import pyplot as plt
fig,axs = plt.subplots(1,3,figsize=(15,5))
axs[0].plot(Xs,Ys0)
axs[1].plot(Xs,Ys1)
axs[2].plot(Xs,Ys2)

# We will use the same network for all three functions
ks = [2.7,8*np.pi,205]
dnn_sizes = [1,102,101,1] # so # of weights is 102*1 + 102*102 + 102*1 = 10608
```

```
fl_sizes = [1,100,100] # so # of weights is  1*a + a*1 + a*a + a*1 = 10300, for a = 100

fl_sizes2 = [1,25,25] # so # of weights is  1*a + a*1 + a*a + a*1 = 700, for a = 25
dnn_sizes2 = [1,26,26,1] # so # of weights is 26*1 + 26*26 + 26*1 = 728
# a three layer nn can represent any multivariate function (continuous or discontinuous) https://arxiv.org/abs/2012.03016
```

205.826229518



[ ]:
```
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
p = np.random.permutation(len(Xs))
Xs = Xs[p]
X = torch.from_numpy(Xs).float().unsqueeze(1).to(device)
split = 0.75
train_X = X[:int(split*len(X))]
test_X = X[int(split*len(X)):]

def update_y(Ys):
    y = torch.from_numpy(Ys[p]).float().unsqueeze(1).to(device)
    train_y = y[:int(split*len(y))]
    test_y = y[int(split*len(y)):]
    return train_y, test_y
train_y,test_y = update_y(Ys0)
```

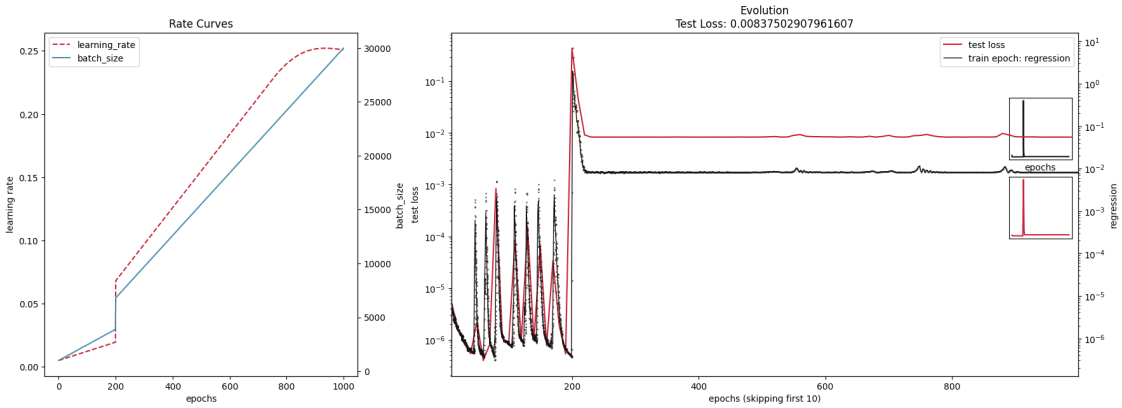
[ ]:
```
D = dojo()
D.epochs=1000
D.max_batch_size=train_X.shape[0]
opt = lambda x: optim.Adam(x, lr=0.00001) # Adam better than SGD and AdamW in quick tests.
```

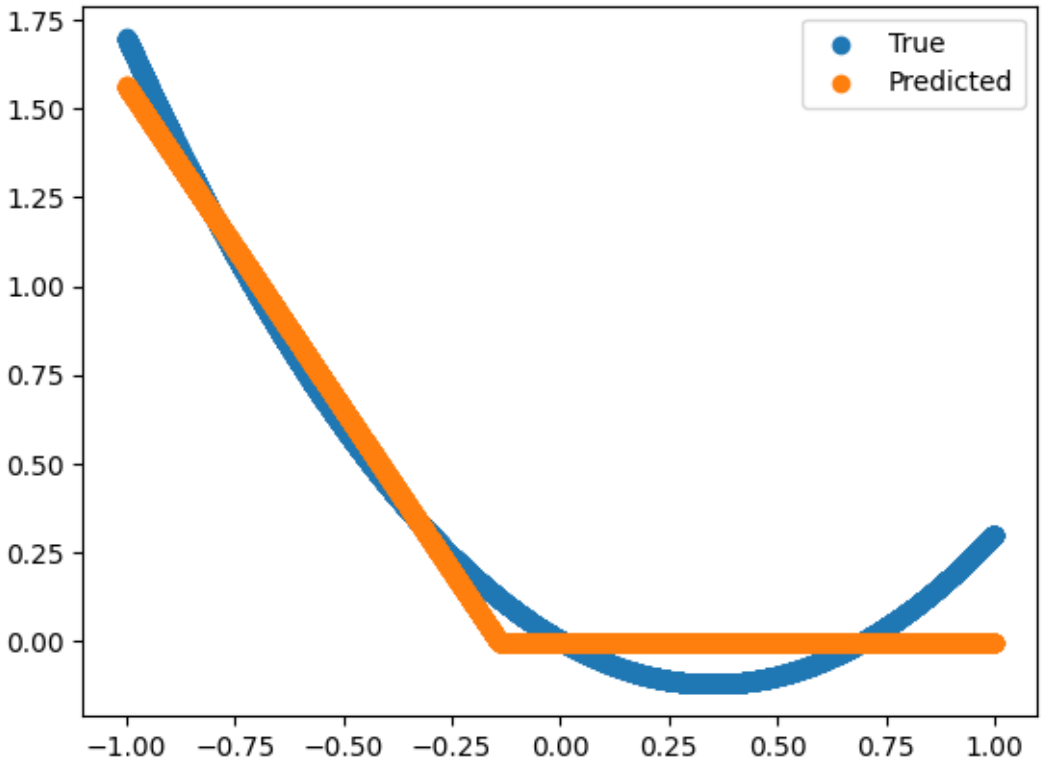
1 Function 0

1.1 DNN 0

[ ]:
```
act = torch.nn.ReLU()
net = DNN(device, dnn_sizes, opttype=opt, act = act, bias=True) # expect 1.6%␣
 ↪error rate
report = D.train(net, train_X, train_y, test_X, test_y, start_batch_size=1000)
ecran(net, test_X, test_y, report, classification=False)
plt.scatter(test_X.detach().cpu().numpy(),test_y.detach().cpu().
 ↪numpy(),label='True')
plt.scatter(test_X.detach().cpu().numpy(),net(test_X).detach().cpu().
 ↪numpy(),label='Predicted')
plt.legend()
plt.show()
```

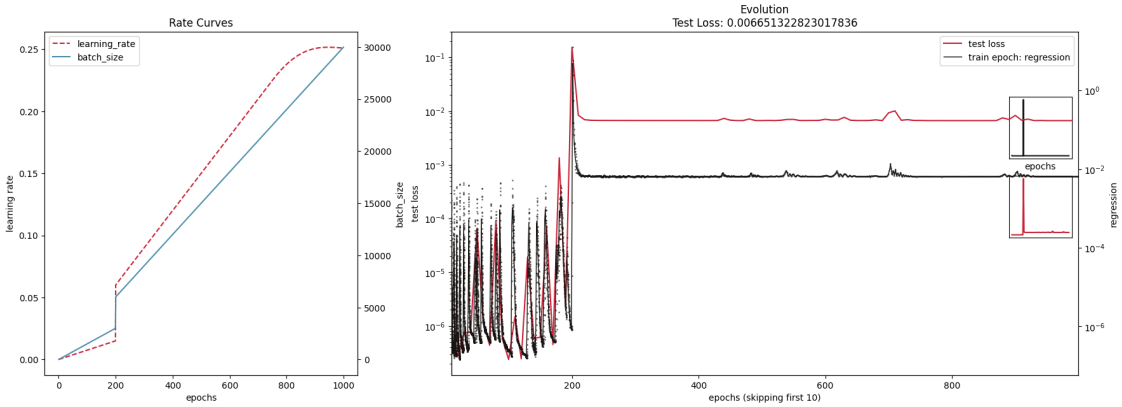
```
100%|      | 1000/1000 [00:12<00:00, 81.81it/s]
```

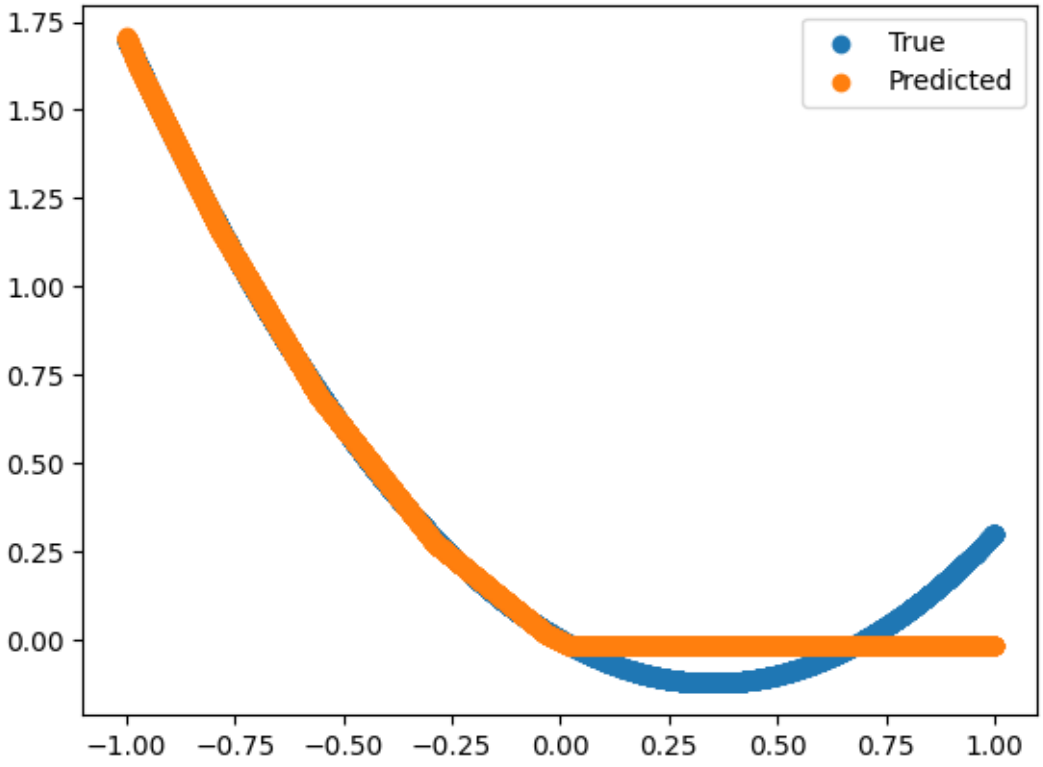




[ ]:
```
act = torch.nn.ReLU()
net = DNN(device, dnn_sizes, opttype=opt, act = act, bias=True) # expect 1.6%␣
 ↪error rate
report = D.train(net, train_X, train_y, test_X, test_y, start_batch_size=10)
ecran(net, test_X, test_y, report, classification=False)
plt.scatter(test_X.detach().cpu().numpy(),test_y.detach().cpu().
 ↪numpy(),label='True')
plt.scatter(test_X.detach().cpu().numpy(),net(test_X).detach().cpu().
 ↪numpy(),label='Predicted')
plt.legend()
plt.show()
```

```
100%|      | 1000/1000 [00:34<00:00, 29.24it/s]
```

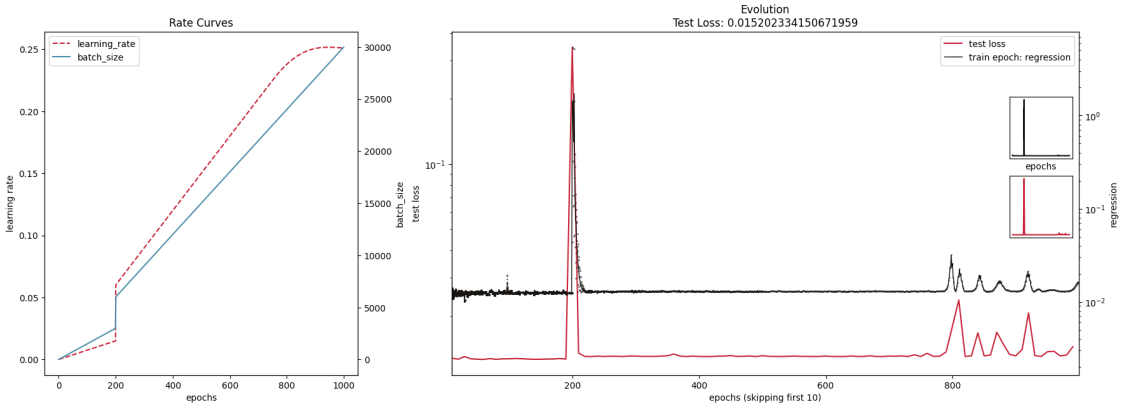


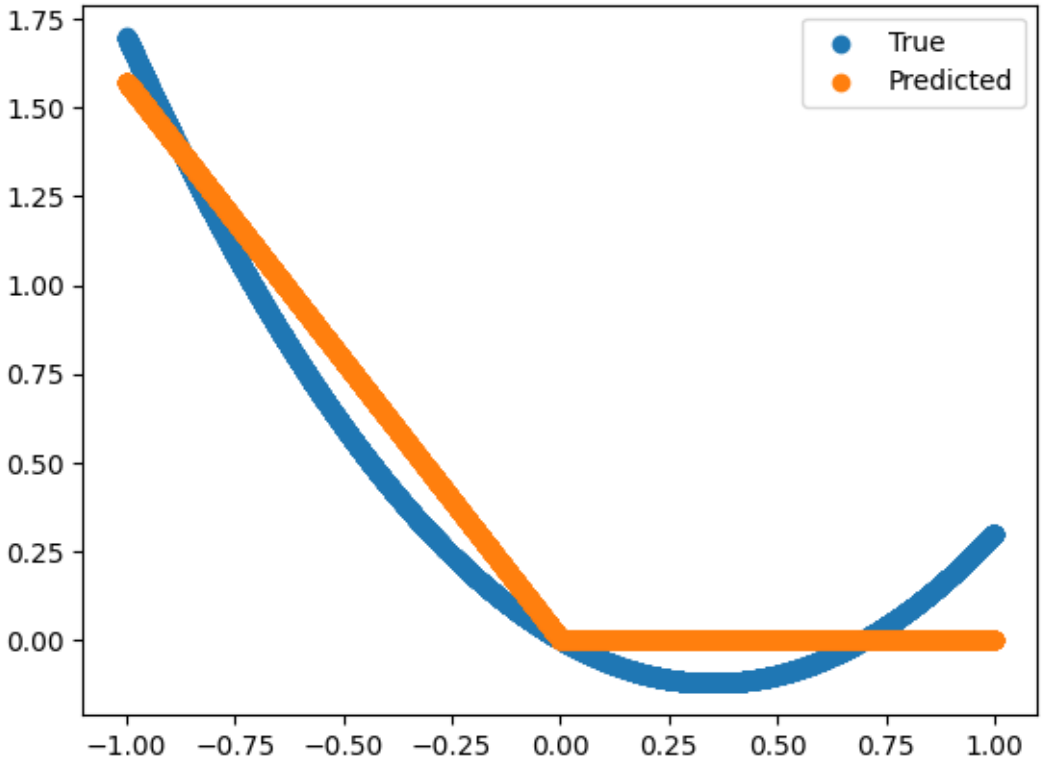


[ ]:
```
act = torch.nn.ReLU()
net = DNN(device, dnn_sizes, opttype=opt, act = act, bias=False) # expect 1.6%␣
 ↪error rate
report = D.train(net, train_X, train_y, test_X, test_y, start_batch_size=10)
ecran(net, test_X, test_y, report, classification=False)
plt.scatter(test_X.detach().cpu().numpy(),test_y.detach().cpu().
 ↪numpy(),label='True')
```

```
plt.scatter(test_X.detach().cpu().numpy(),net(test_X).detach().cpu().
  ↪numpy(),label='Predicted')
plt.legend()
plt.show()
```

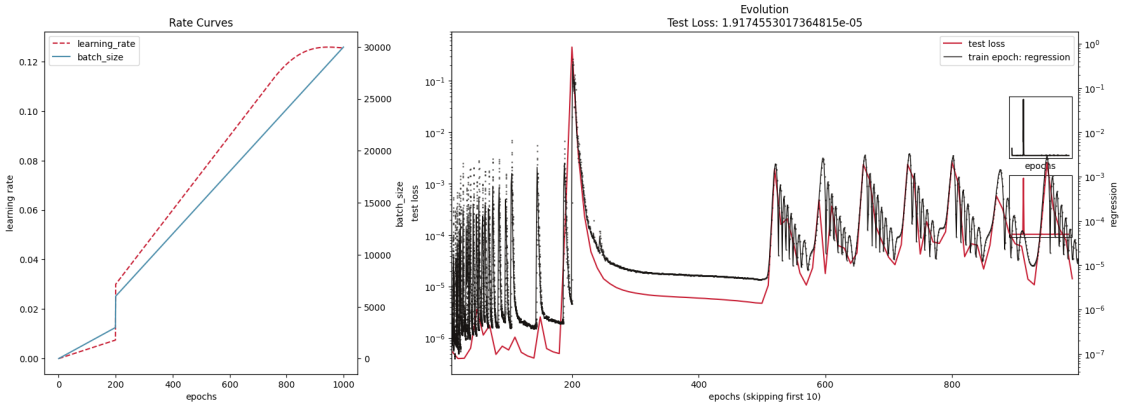
```
100%|      | 1000/1000 [00:23<00:00, 42.29it/s]
```

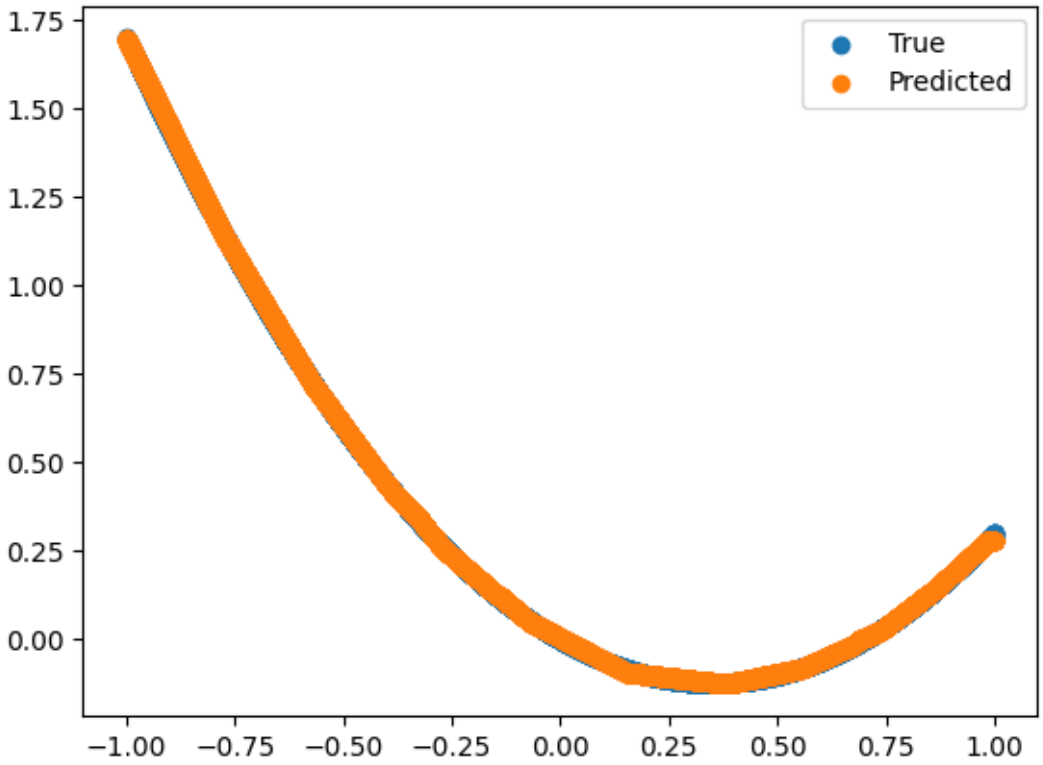




[ ]:
```
act = torch.nn.ReLU()
opt2 = lambda x: optim.Adam(x, lr=0.000005)
net = DNN(device, dnn_sizes, opttype=opt2, act = act, bias=True) # expect 1.6%␣
 ↪error rate
report = D.train(net, train_X, train_y, test_X, test_y, start_batch_size=10,␣
 ↪max_batch_size=100)
ecran(net, test_X, test_y, report, classification=False)
plt.scatter(test_X.detach().cpu().numpy(),test_y.detach().cpu().
 ↪numpy(),label='True')
plt.scatter(test_X.detach().cpu().numpy(),net(test_X).detach().cpu().
 ↪numpy(),label='Predicted')
plt.legend()
plt.show()
```

```
100%|      | 1000/1000 [00:37<00:00, 26.85it/s]
```

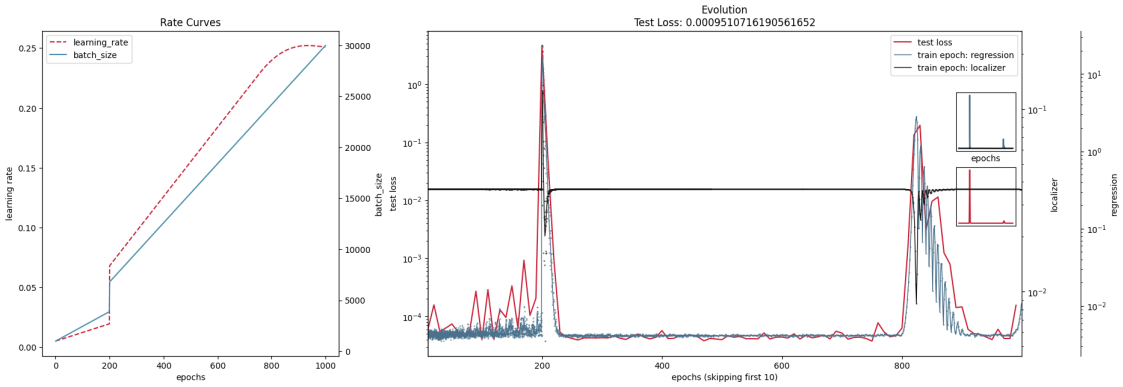


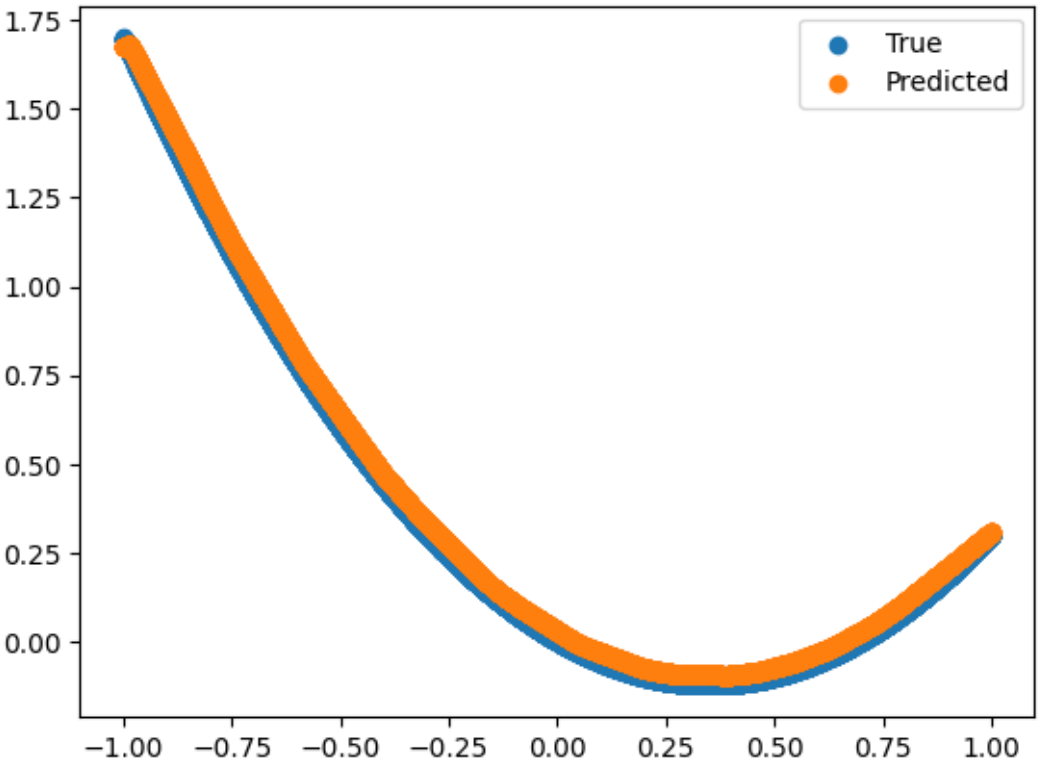


1.2 FL 0

[ ]:
```
delta = np.array([1]*len(fl_sizes))*0.05
k = [ks[0]]*len(fl_sizes)
act = torch.nn.ReLU()
net = FL(device, fl_sizes, delta, k, opttype=opt, act = act, bias=True) #␣
 ↪expect 1.6% error rate
report = D.train(net, train_X, train_y, test_X, test_y, start_batch_size=1000)
ecran(net, test_X, test_y, report, classification=False)
plt.scatter(test_X.detach().cpu().numpy(),test_y.detach().cpu().
 ↪numpy(),label='True')
plt.scatter(test_X.detach().cpu().numpy(),net(test_X).detach().cpu().
 ↪numpy(),label='Predicted')
plt.legend()
plt.show()
```

```
100%|      | 1000/1000 [00:45<00:00, 22.14it/s]
```

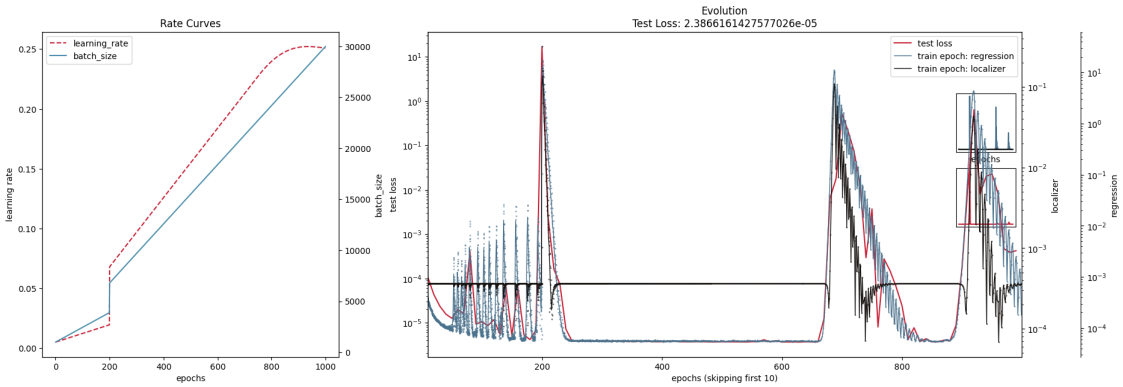


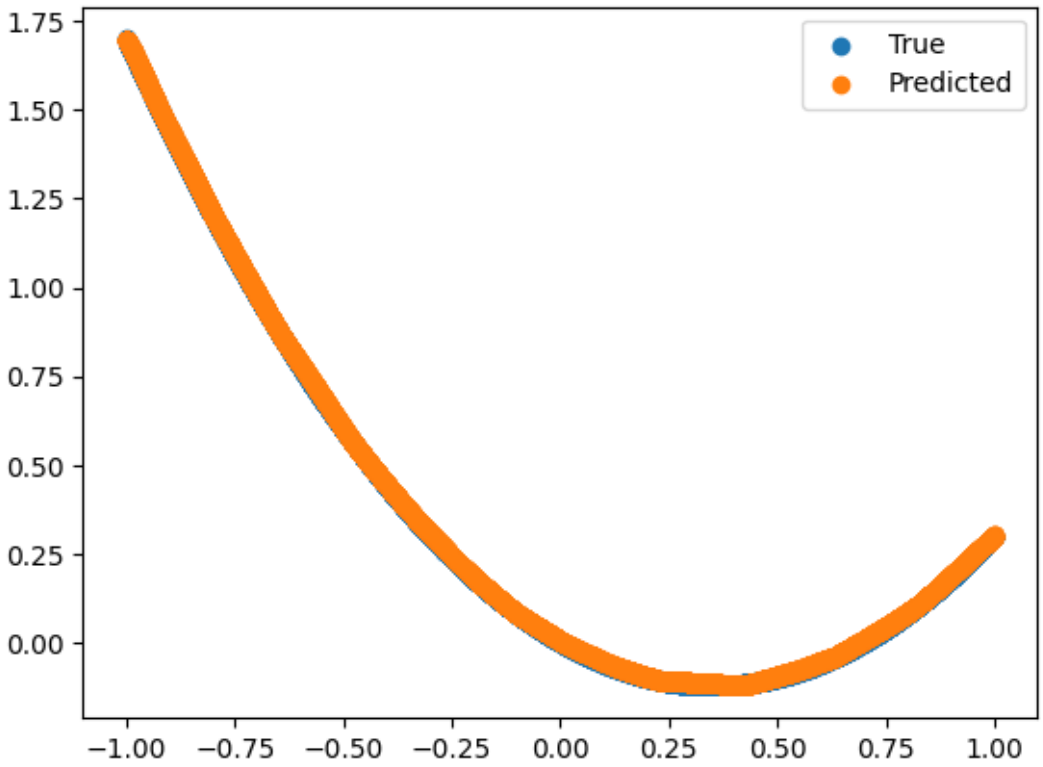



```
[ ]: delta = np.array([1]*len(fl_sizes))*0.005
     k = [ks[0]]*len(fl_sizes)
     act = torch.nn.ReLU()
     net = FL(device, fl_sizes, delta, k, opttype=opt, act = act, bias=True) #␣
       ↪expect 1.6% error rate
     report = D.train(net, train_X, train_y, test_X, test_y, start_batch_size=1000)
     ecran(net, test_X, test_y, report, classification=False)
```

```
plt.scatter(test_X.detach().cpu().numpy(),test_y.detach().cpu().
  ↪numpy(),label='True')
plt.scatter(test_X.detach().cpu().numpy(),net(test_X).detach().cpu().
  ↪numpy(),label='Predicted')
plt.legend()
plt.show()
```

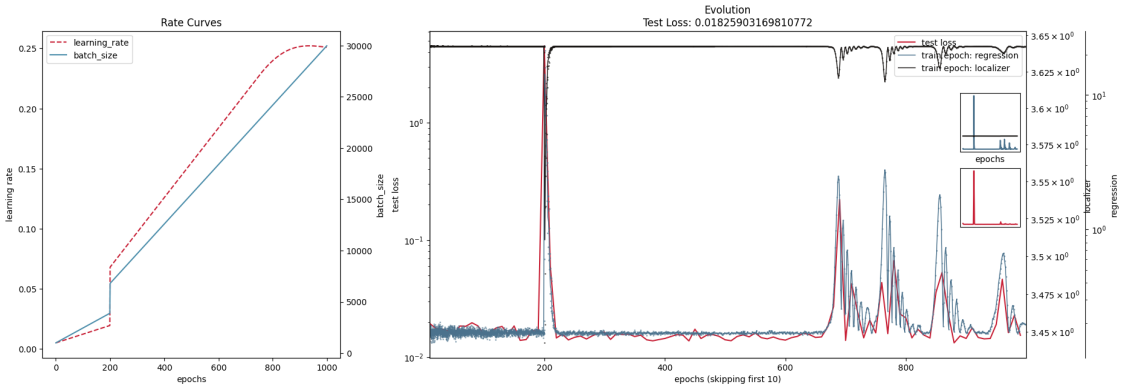
```
100%|      | 1000/1000 [00:43<00:00, 23.22it/s]
```

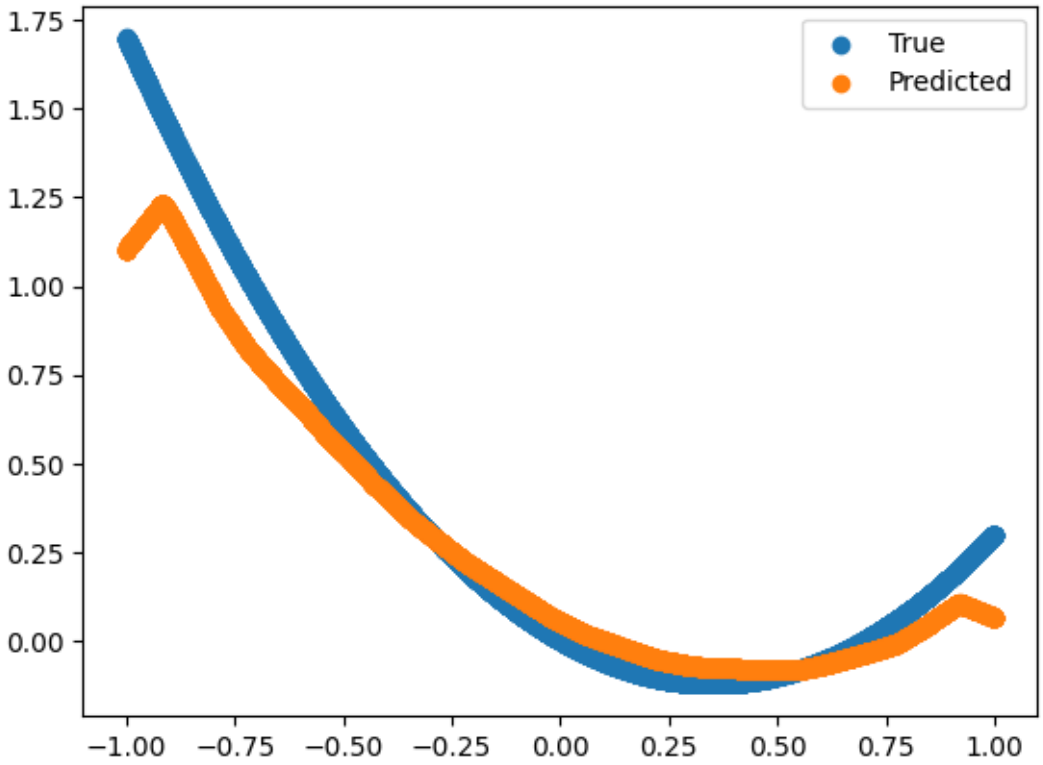




[ ]:
```
delta = np.array([1]*len(fl_sizes))*0.5
k = [ks[0]]*len(fl_sizes)
act = torch.nn.ReLU()
net = FL(device, fl_sizes, delta, k, opttype=opt, act = act, bias=True) #␣
 ↪expect 1.6% error rate
report = D.train(net, train_X, train_y, test_X, test_y, start_batch_size=1000)
ecran(net, test_X, test_y, report, classification=False)
plt.scatter(test_X.detach().cpu().numpy(),test_y.detach().cpu().
 ↪numpy(),label='True')
plt.scatter(test_X.detach().cpu().numpy(),net(test_X).detach().cpu().
 ↪numpy(),label='Predicted')
plt.legend()
plt.show()
```

```
100%|      | 1000/1000 [00:43<00:00, 22.89it/s]
```



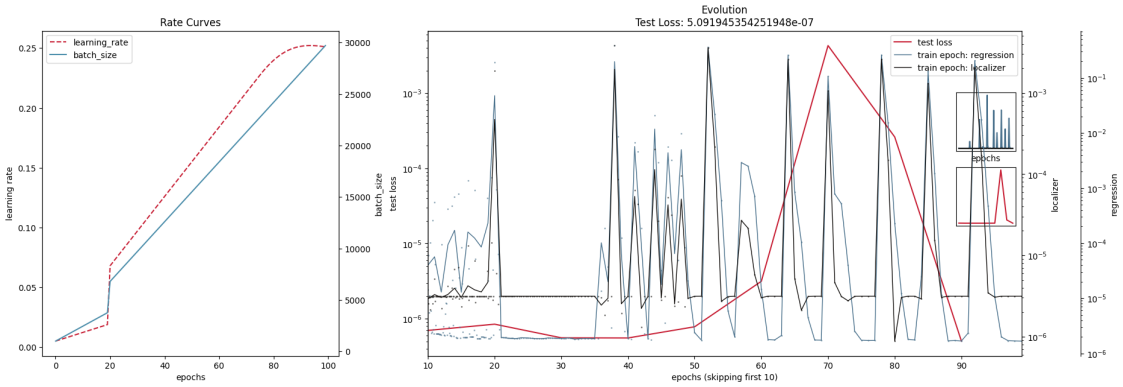


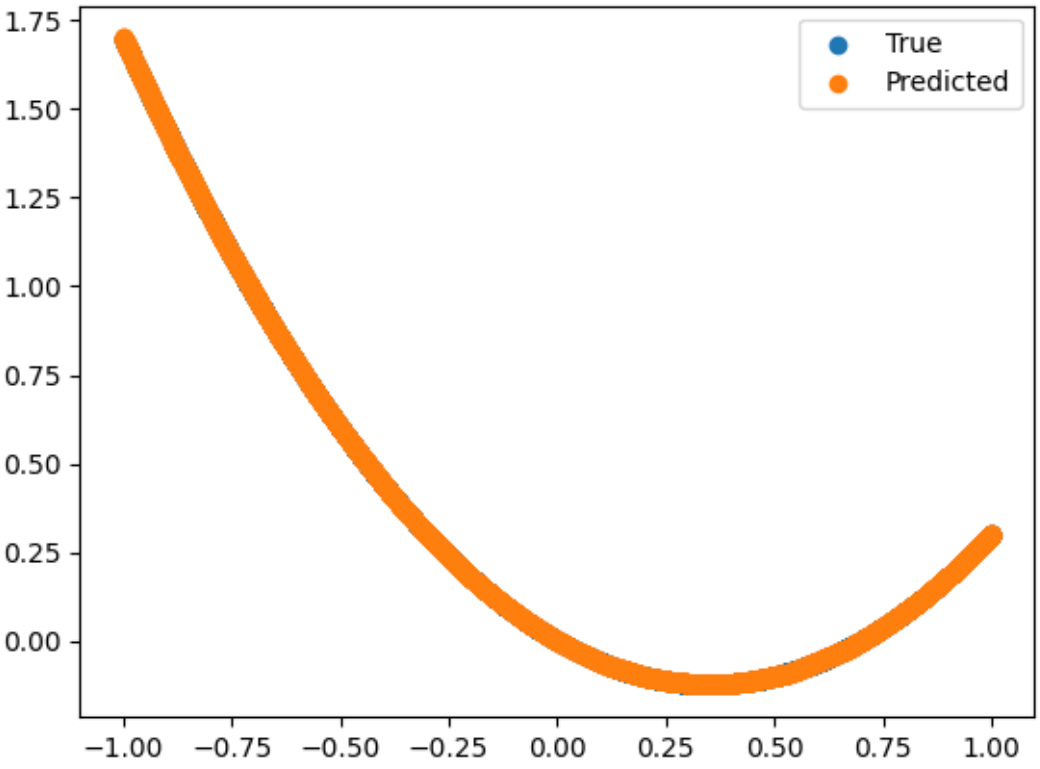
[ ]:

1.3 FL 0 Greedy

[ ]:
```
act = torch.nn.ReLU()
delta = np.array([1]*len(fl_sizes))*0.00005
k = [ks[1]]*len(fl_sizes)
net = FL(device, fl_sizes, delta, k, opttype=opt, act = act, bias=True) #␣
 ↪expect 1.6% error rate
D.epochs = 100
report = D.train(net, train_X, train_y, test_X, test_y, start_batch_size=1000,␣
 ↪repeat_epochs=40)
D.epochs = 1000
ecran(net, test_X, test_y, report, classification=False)
plt.scatter(test_X.detach().cpu().numpy(),test_y.detach().cpu().
 ↪numpy(),label='True')
plt.scatter(test_X.detach().cpu().numpy(),net(test_X).detach().cpu().
 ↪numpy(),label='Predicted')
plt.legend()
plt.show()
```

```
100%|      | 100/100 [02:55<00:00,  1.75s/it]
```





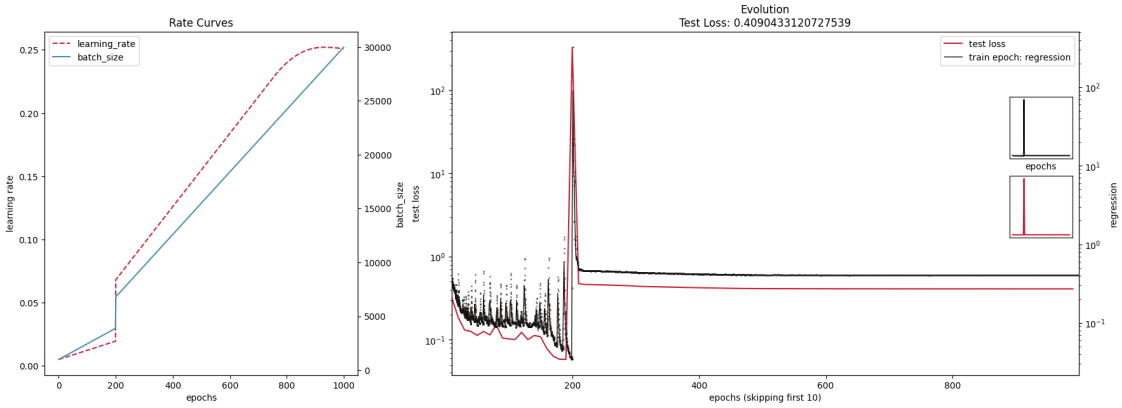
2 Function 1

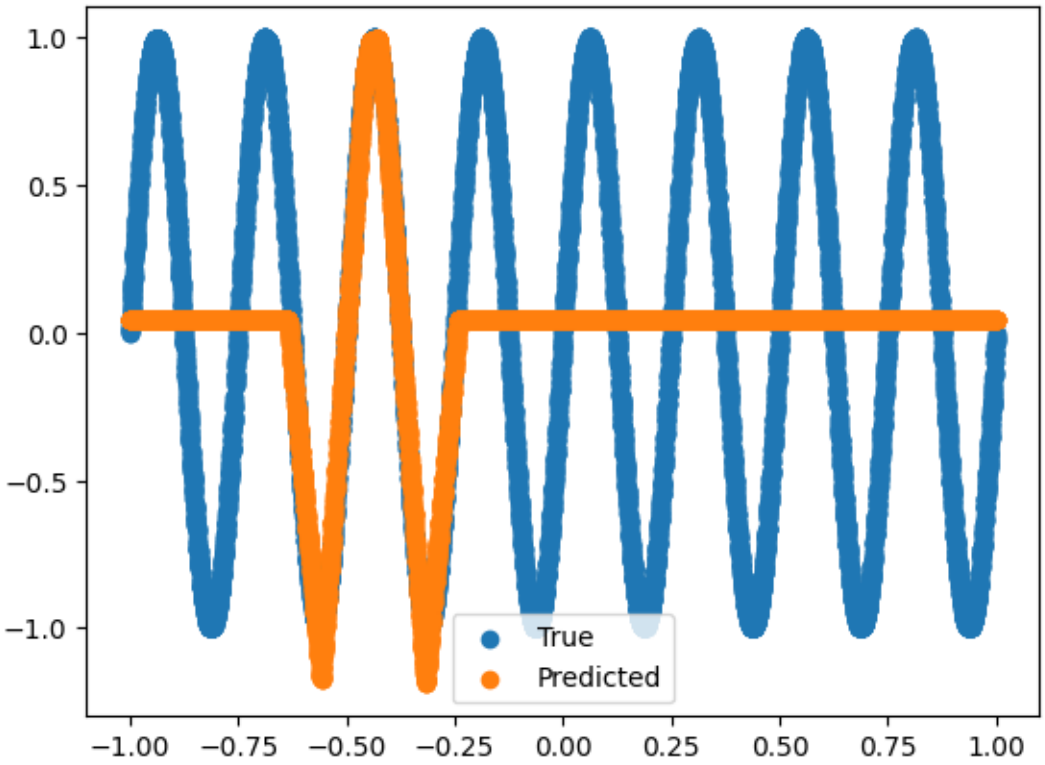
[ ]:
```
train_y, test_y = update_y(Ys1)
```

2.1 DNN 1

[ ]:
```
act = torch.nn.ReLU()
net = DNN(device, dnn_sizes, opttype=opt, act = act, bias=True) # expect 1.6%␣
 ↪error rate
report = D.train(net, train_X, train_y, test_X, test_y, start_batch_size=1000)
ecran(net, test_X, test_y, report, classification=False)
plt.scatter(test_X.detach().cpu().numpy(),test_y.detach().cpu().
 ↪numpy(),label='True')
plt.scatter(test_X.detach().cpu().numpy(),net(test_X).detach().cpu().
 ↪numpy(),label='Predicted')
plt.legend()
plt.show()
```

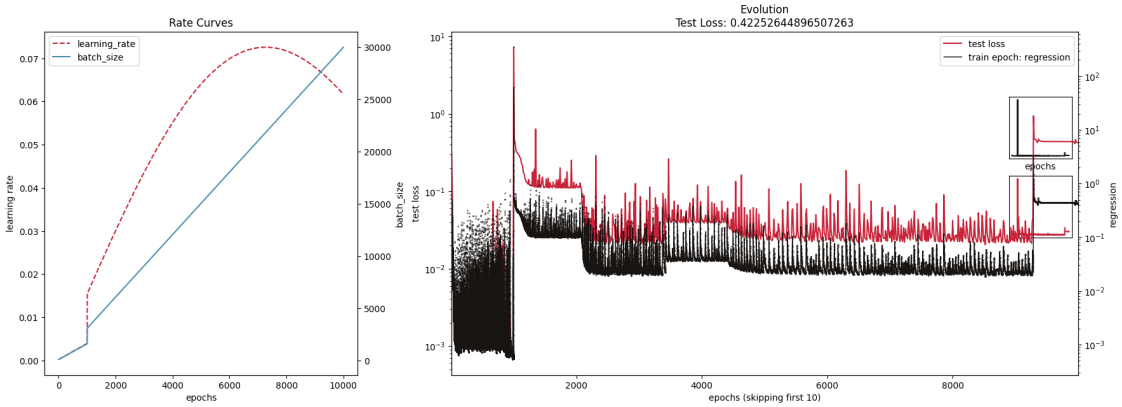
```
100%|      | 1000/1000 [00:11<00:00, 85.92it/s]
```

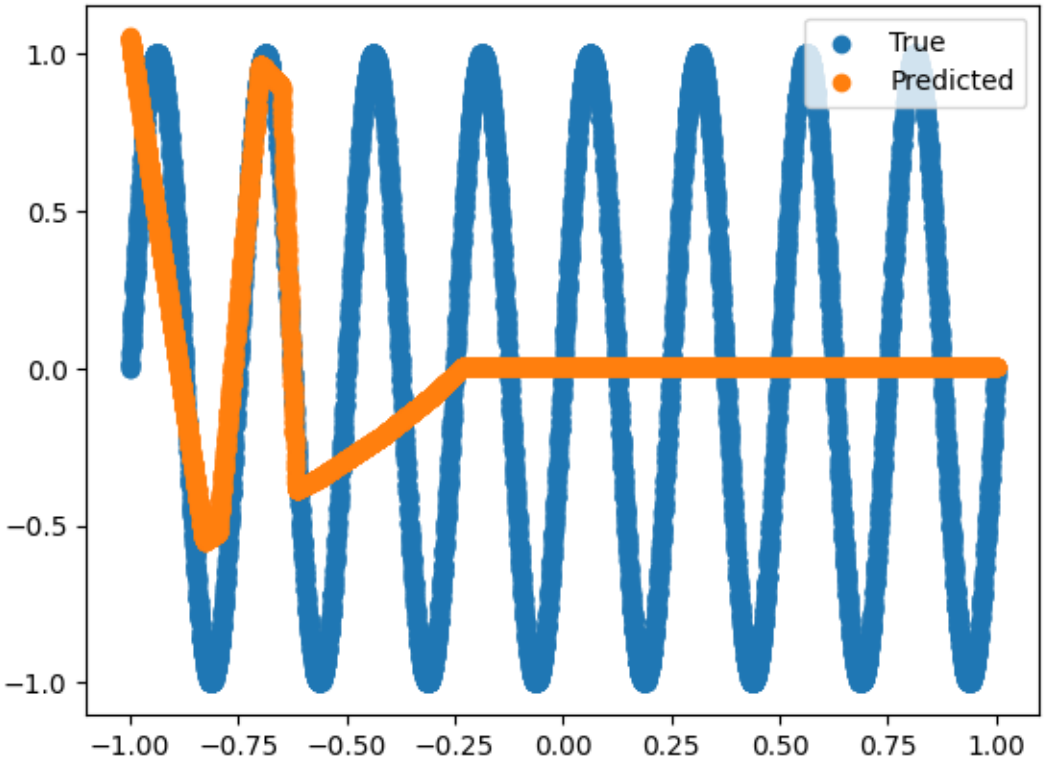




[ ]:
```
opt2 = lambda x: optim.Adam(x, lr=0.000005)
act = torch.nn.ReLU()
net = DNN(device, dnn_sizes, opttype=opt2, act = act, bias=True) # expect 1.6%␣
 ↪error rate
D.epochs = 10000
report = D.train(net, train_X, train_y, test_X, test_y, start_batch_size=100,␣
 ↪max_batch_size=100, splits =[0.1,0.1], final_percent_lr=0.1)
D.epochs = 1000
ecran(net, test_X, test_y, report, classification=False)
plt.scatter(test_X.detach().cpu().numpy(),test_y.detach().cpu().
 ↪numpy(),label='True')
plt.scatter(test_X.detach().cpu().numpy(),net(test_X).detach().cpu().
 ↪numpy(),label='Predicted')
plt.legend()
plt.show()
```

```
100%|      | 10000/10000 [03:15<00:00, 51.10it/s]
```



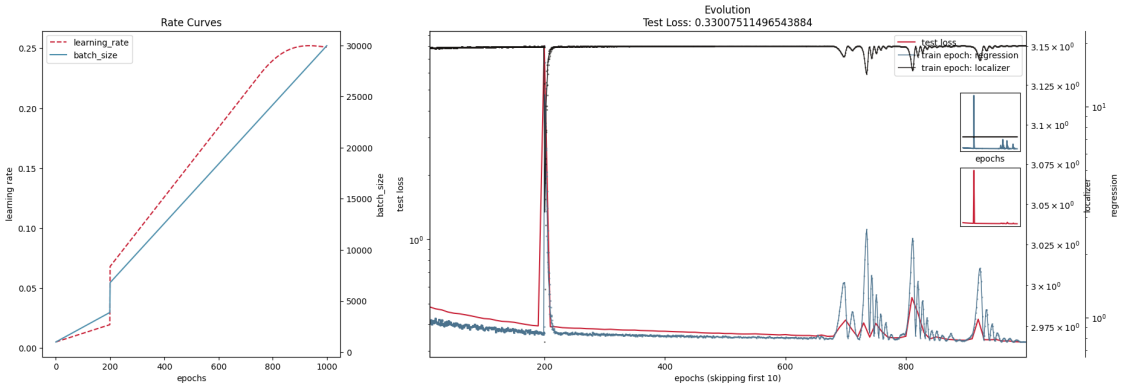


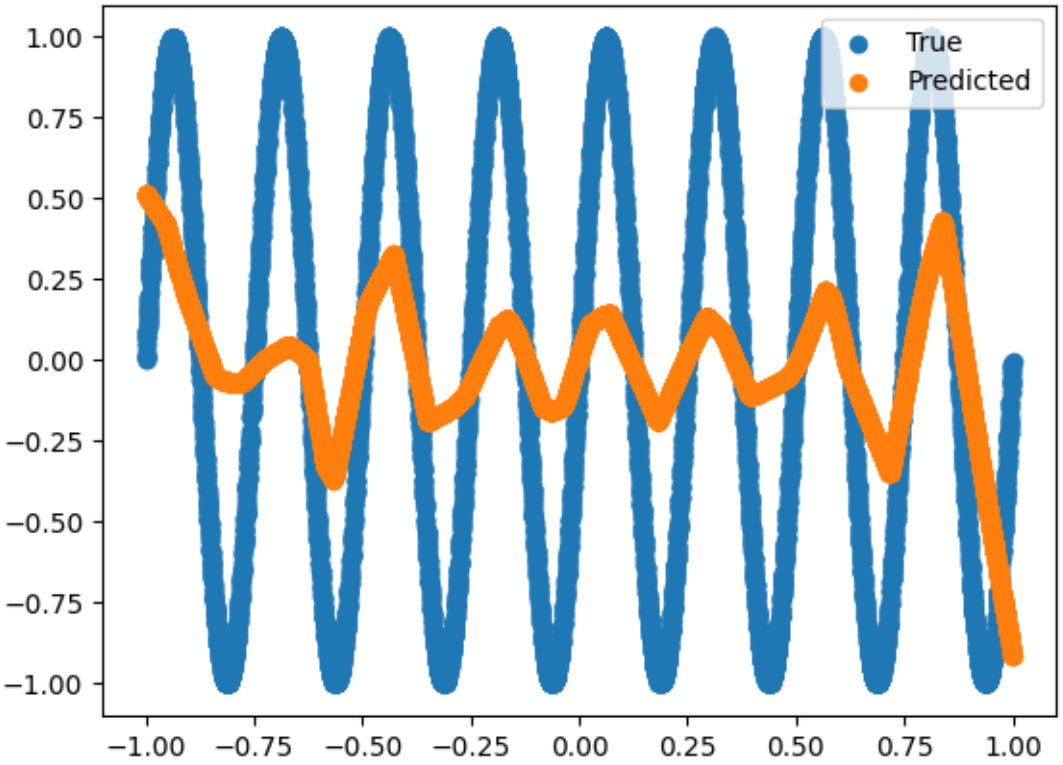
2.2 FL 1

[ ]:
```
act = torch.nn.ReLU()
delta = np.array([1]*len(fl_sizes))*0.05
k = [ks[1]]*len(fl_sizes)
net = FL(device, fl_sizes, delta, k, opttype=opt, act = act, bias=True) #␣
 ↪expect 1.6% error rate
```

```
report = D.train(net, train_X, train_y, test_X, test_y, start_batch_size=1000)
ecran(net, test_X, test_y, report, classification=False)
plt.scatter(test_X.detach().cpu().numpy(),test_y.detach().cpu().
 ↪numpy(),label='True')
plt.scatter(test_X.detach().cpu().numpy(),net(test_X).detach().cpu().
 ↪numpy(),label='Predicted')
plt.legend()
plt.show()
```

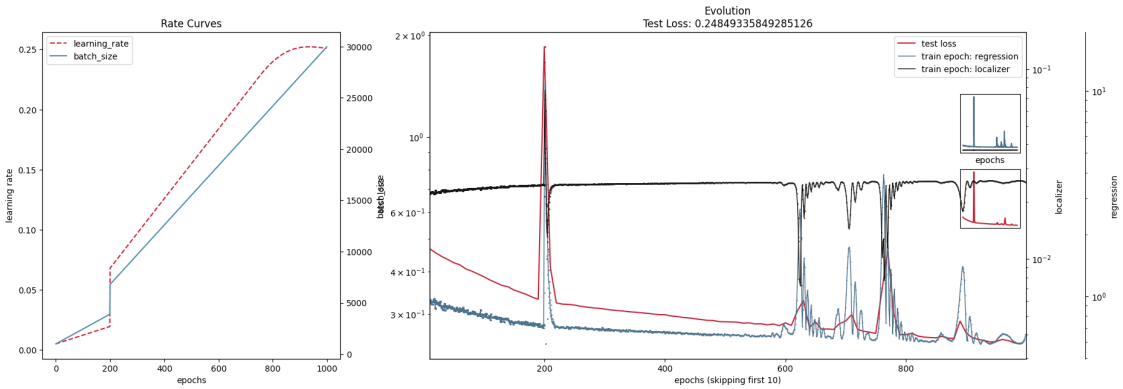
```
100%|      | 1000/1000 [00:42<00:00, 23.80it/s]
```

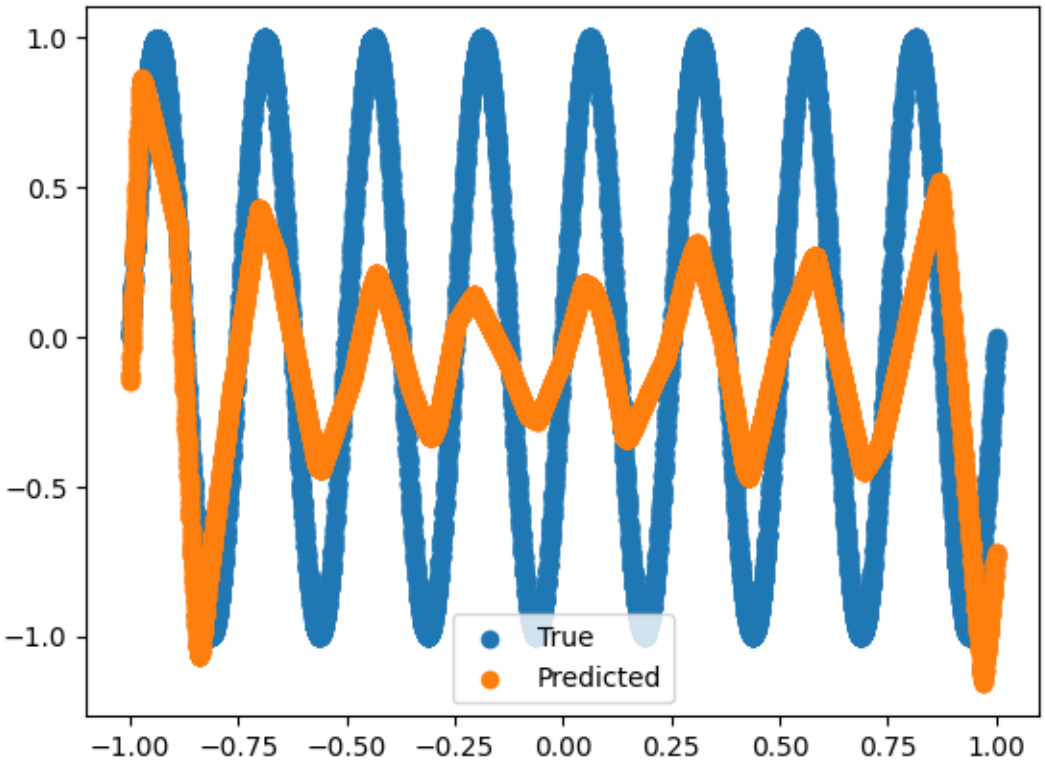




[ ]:
```
act = torch.nn.ReLU()
delta = np.array([1]*len(fl_sizes))*0.005
k = [ks[1]]*len(fl_sizes)
net = FL(device, fl_sizes, delta, k, opttype=opt, act = act, bias=True) #␣
 ↪expect 1.6% error rate
report = D.train(net, train_X, train_y, test_X, test_y, start_batch_size=1000)
ecran(net, test_X, test_y, report, classification=False)
plt.scatter(test_X.detach().cpu().numpy(),test_y.detach().cpu().
 ↪numpy(),label='True')
plt.scatter(test_X.detach().cpu().numpy(),net(test_X).detach().cpu().
 ↪numpy(),label='Predicted')
plt.legend()
plt.show()
```

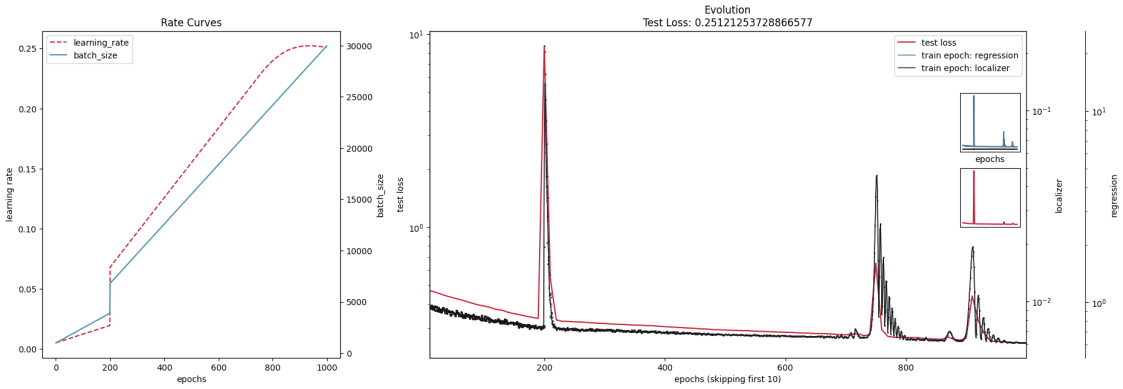
```
100%|      | 1000/1000 [00:43<00:00, 23.15it/s]
```

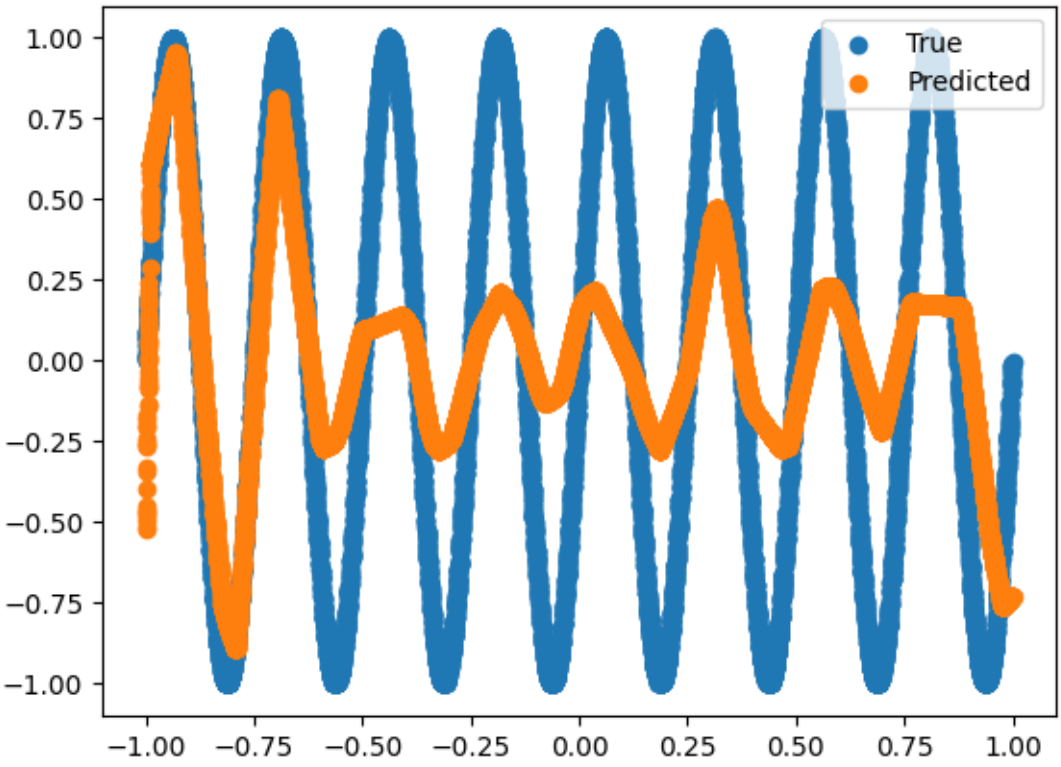




[ ]:
```
act = torch.nn.ReLU()
delta = np.array([1]*len(fl_sizes))*0.00005
k = [ks[1]]*len(fl_sizes)
net = FL(device, fl_sizes, delta, k, opttype=opt, act = act, bias=True) #␣
 ↪expect 1.6% error rate
report = D.train(net, train_X, train_y, test_X, test_y, start_batch_size=1000)
ecran(net, test_X, test_y, report, classification=False)
plt.scatter(test_X.detach().cpu().numpy(),test_y.detach().cpu().
 ↪numpy(),label='True')
plt.scatter(test_X.detach().cpu().numpy(),net(test_X).detach().cpu().
 ↪numpy(),label='Predicted')
plt.legend()
plt.show()
```

```
100%|      | 1000/1000 [00:45<00:00, 21.94it/s]
```

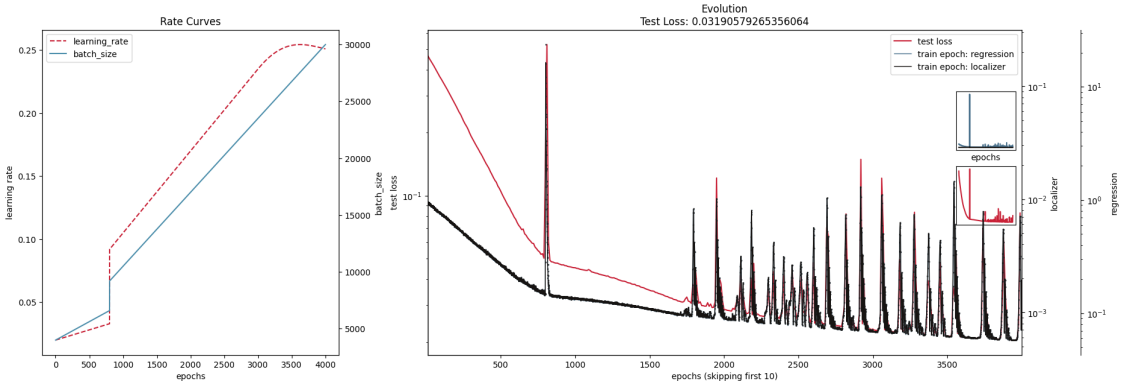


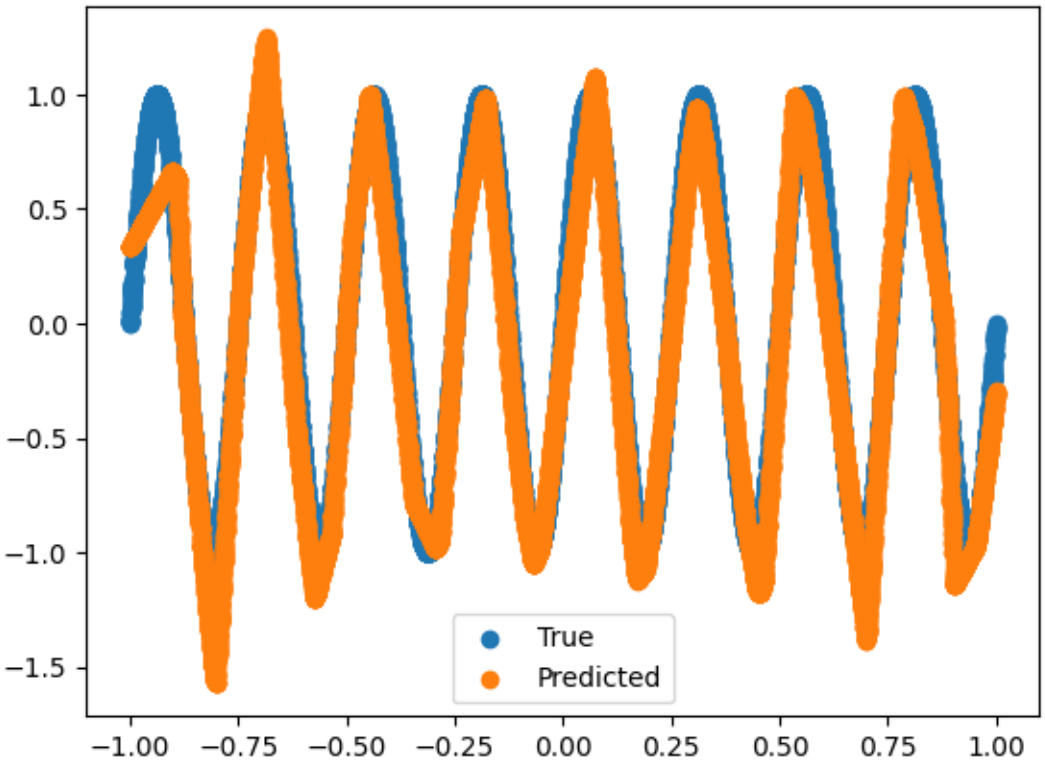


[ ]:
```
act = torch.nn.ReLU()
delta = np.array([1]*len(fl_sizes))*0.00005
k = [ks[1]]*len(fl_sizes)
net = FL(device, fl_sizes, delta, k, opttype=opt, act = act, bias=True) #␣
 ↪expect 1.6% error rate
D.epochs = 4000
report = D.train(net, train_X, train_y, test_X, test_y, start_batch_size=4000)
D.epochs = 1000
```

```
ecran(net, test_X, test_y, report, classification=False)
plt.scatter(test_X.detach().cpu().numpy(),test_y.detach().cpu().
 ↪numpy(),label='True')
plt.scatter(test_X.detach().cpu().numpy(),net(test_X).detach().cpu().
 ↪numpy(),label='Predicted')
plt.legend()
plt.show()
```

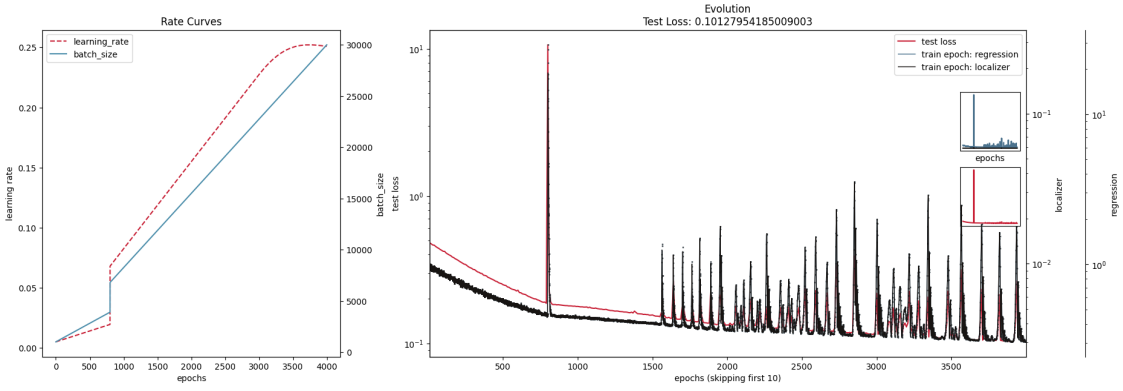
```
100%|      | 4000/4000 [02:12<00:00, 30.12it/s]
```

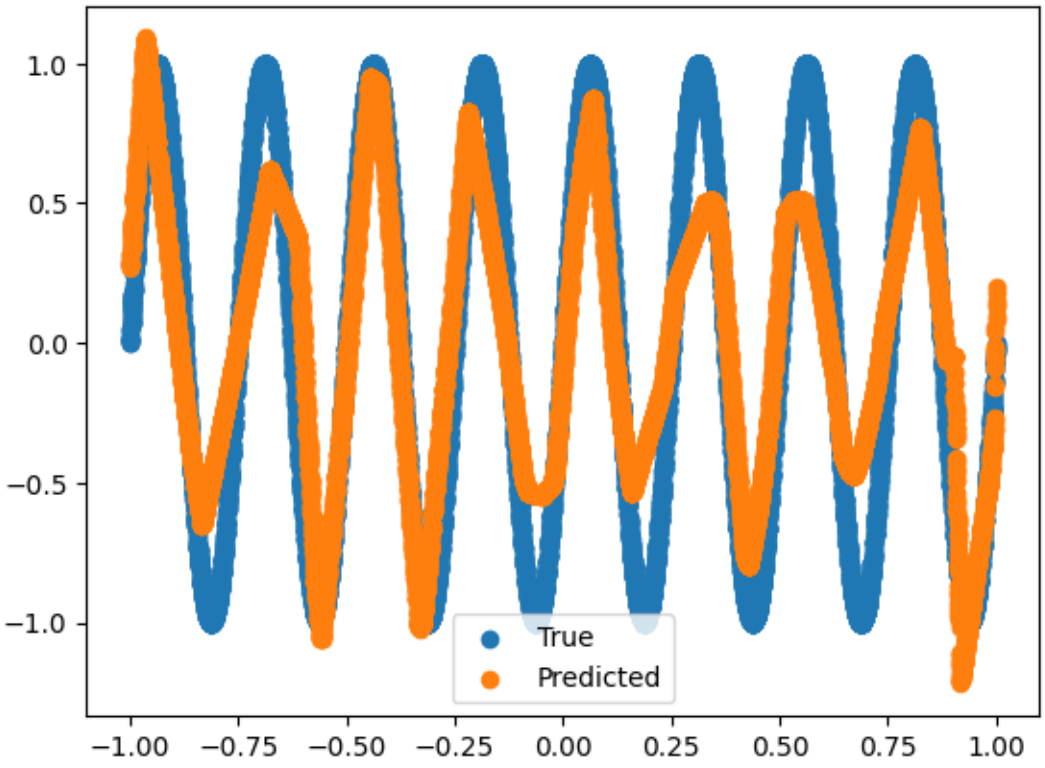




[ ]:
```
act = torch.nn.ReLU()
delta = np.array([1]*len(fl_sizes))*0.00005
k = [ks[1]]*len(fl_sizes)
net = FL(device, fl_sizes, delta, k, opttype=opt, act = act, bias=True) #␣
 ↪expect 1.6% error rate
D.epochs = 4000
report = D.train(net, train_X, train_y, test_X, test_y, start_batch_size=1000)
D.epochs = 1000
ecran(net, test_X, test_y, report, classification=False)
plt.scatter(test_X.detach().cpu().numpy(),test_y.detach().cpu().
 ↪numpy(),label='True')
plt.scatter(test_X.detach().cpu().numpy(),net(test_X).detach().cpu().
 ↪numpy(),label='Predicted')
plt.legend()
plt.show()
```

```
100%|      | 4000/4000 [03:04<00:00, 21.71it/s]
```

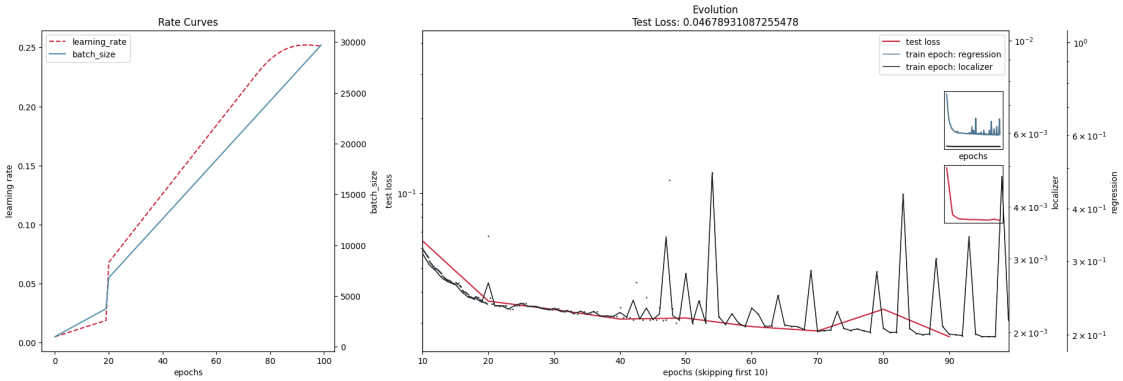


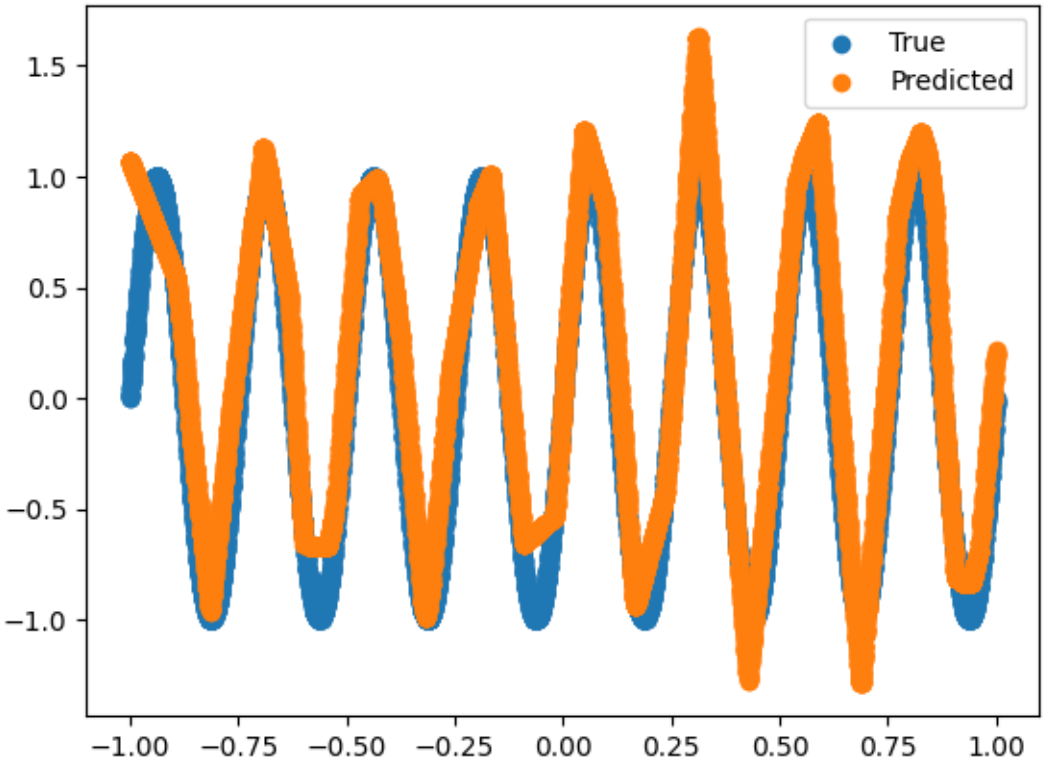


2.3 FL 1 Greedy

[ ]:
```
act = torch.nn.ReLU()
delta = np.array([1]*len(fl_sizes))*0.00005
k = [ks[1]]*len(fl_sizes)
net = FL(device, fl_sizes, delta, k, opttype=opt, act = act, bias=True) #␣
 ↪expect 1.6% error rate
D.epochs = 100
report = D.train(net, train_X, train_y, test_X, test_y, start_batch_size=1000,␣
 ↪repeat_epochs=40)
D.epochs = 1000
ecran(net, test_X, test_y, report, classification=False)
plt.scatter(test_X.detach().cpu().numpy(),test_y.detach().cpu().
 ↪numpy(),label='True')
plt.scatter(test_X.detach().cpu().numpy(),net(test_X).detach().cpu().
 ↪numpy(),label='Predicted')
plt.legend()
plt.show()
```

```
100%|      | 100/100 [03:02<00:00,  1.82s/it]
```





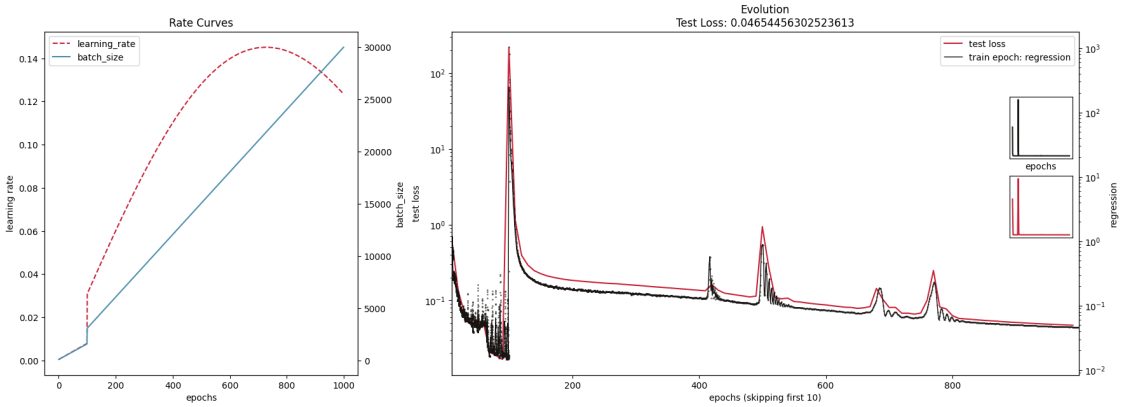
3 Function 2

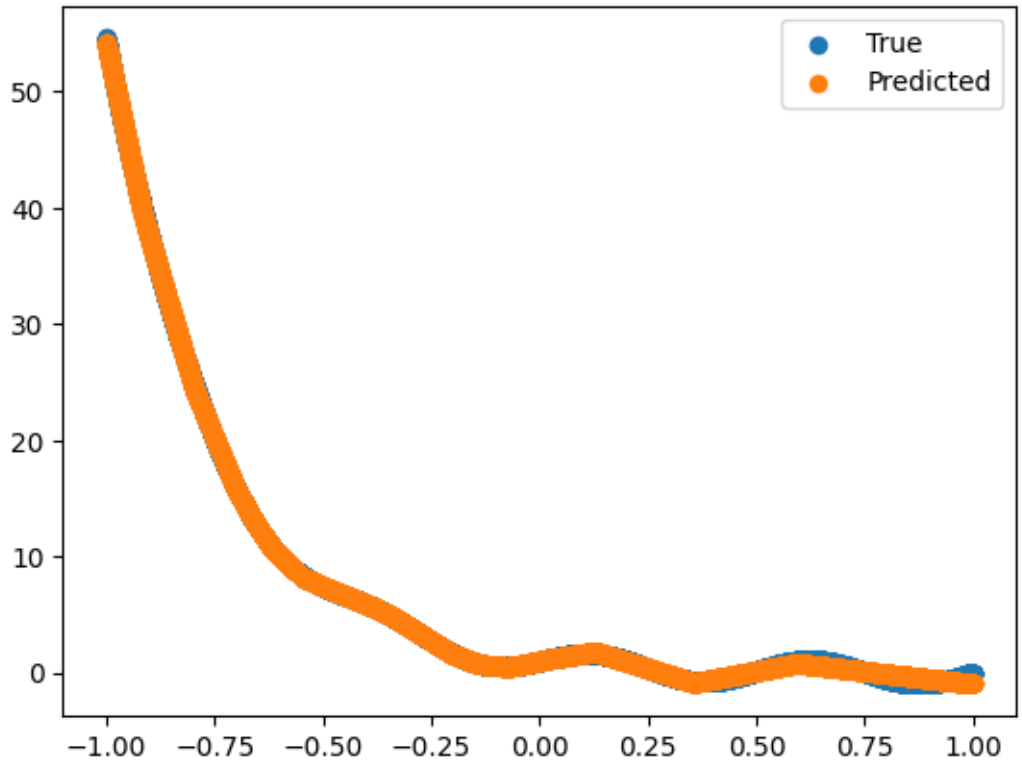
```
[ ]: train_y, test_y = update_y(Ys2)
```

3.1 DNN 2

[ ]:
```
net = DNN(device, dnn_sizes2, opttype=opt, act = act, bias=True) # expect 1.6%␣
 ↪error rate
report = D.train(net, train_X, train_y, test_X, test_y, start_batch_size=100,␣
 ↪max_batch_size=100, splits =[0.1,0.1], final_percent_lr=0.1)
ecran(net, test_X, test_y, report, classification=False)
plt.scatter(test_X.detach().cpu().numpy(),test_y.detach().cpu().
 ↪numpy(),label='True')
plt.scatter(test_X.detach().cpu().numpy(),net(test_X).detach().cpu().
 ↪numpy(),label='Predicted')
plt.legend()
plt.show()
```

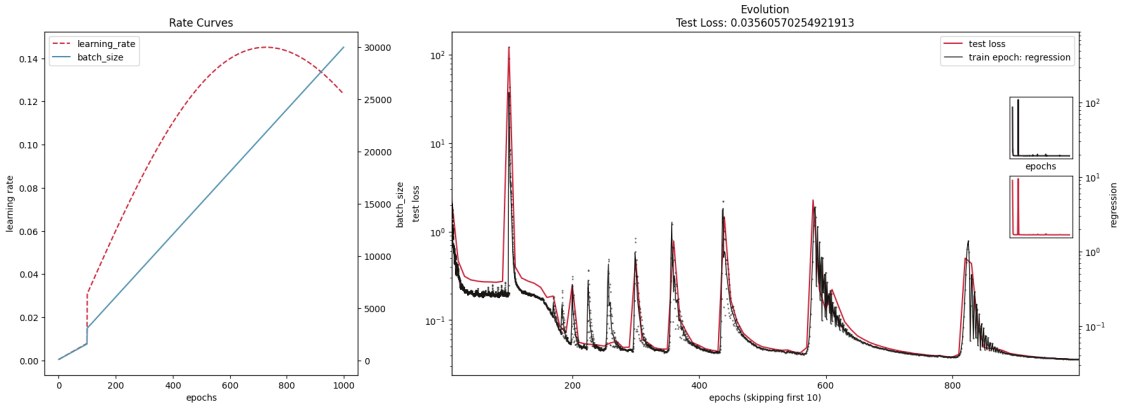
```
100%|      | 1000/1000 [00:17<00:00, 55.89it/s]
```

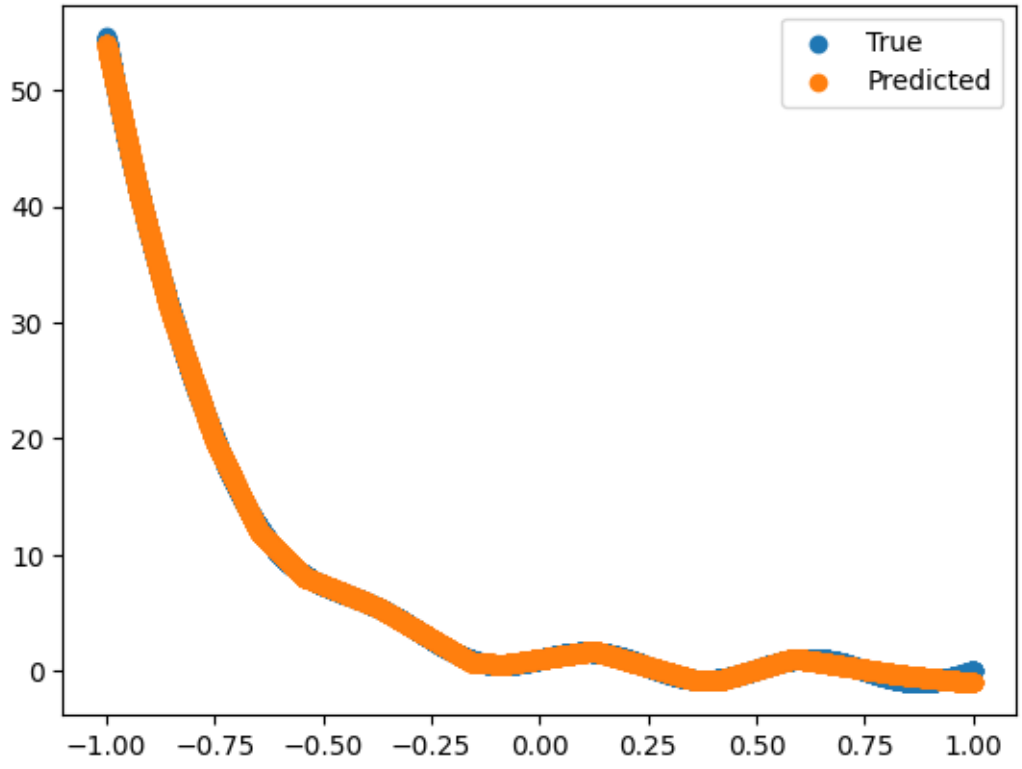




[ ]:
```
net = DNN(device, dnn_sizes2, opttype=opt, act = act, bias=True) # expect 1.6%␣
 ↪error rate
report = D.train(net, train_X, train_y, test_X, test_y, start_batch_size=100,␣
 ↪splits =[0.1,0.1], final_percent_lr=0.1)
ecran(net, test_X, test_y, report, classification=False)
plt.scatter(test_X.detach().cpu().numpy(),test_y.detach().cpu().
 ↪numpy(),label='True')
plt.scatter(test_X.detach().cpu().numpy(),net(test_X).detach().cpu().
 ↪numpy(),label='Predicted')
plt.legend()
plt.show()
```

```
100%|      | 1000/1000 [00:22<00:00, 44.51it/s]
```

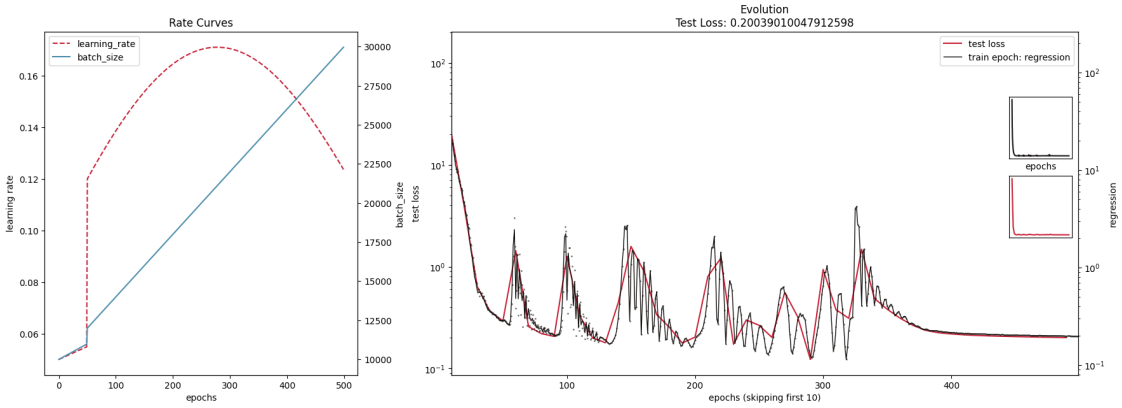


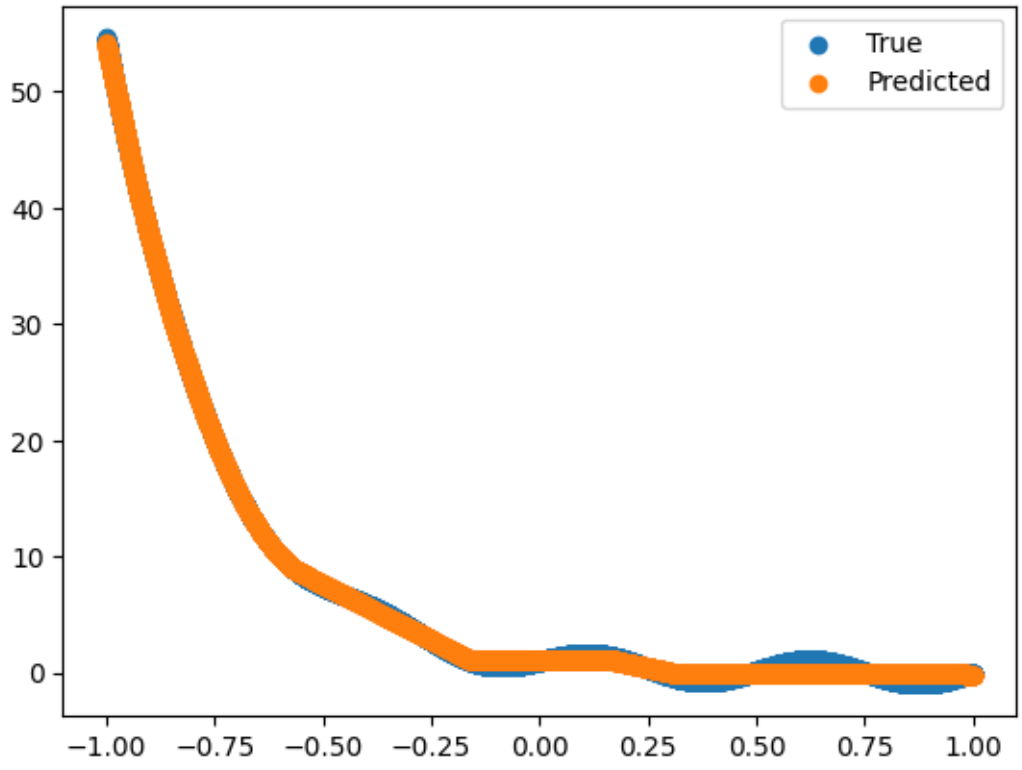


[ ]:
```
net = DNN(device, dnn_sizes2, opttype=opt, act = act, bias=True) # expect 1.6%␣
  ↪error rate
D.epochs=500
report = D.train(net, train_X, train_y, test_X, test_y, start_batch_size=10000,␣
  ↪splits =[0.1,0.1], final_percent_lr=0.1)
D.epochs=1000
ecran(net, test_X, test_y, report, classification=False)
```

```
plt.scatter(test_X.detach().cpu().numpy(),test_y.detach().cpu().
  ↪numpy(),label='True')
plt.scatter(test_X.detach().cpu().numpy(),net(test_X).detach().cpu().
  ↪numpy(),label='Predicted')
plt.legend()
plt.show()
```

```
100%|      | 500/500 [00:02<00:00, 216.71it/s]
```

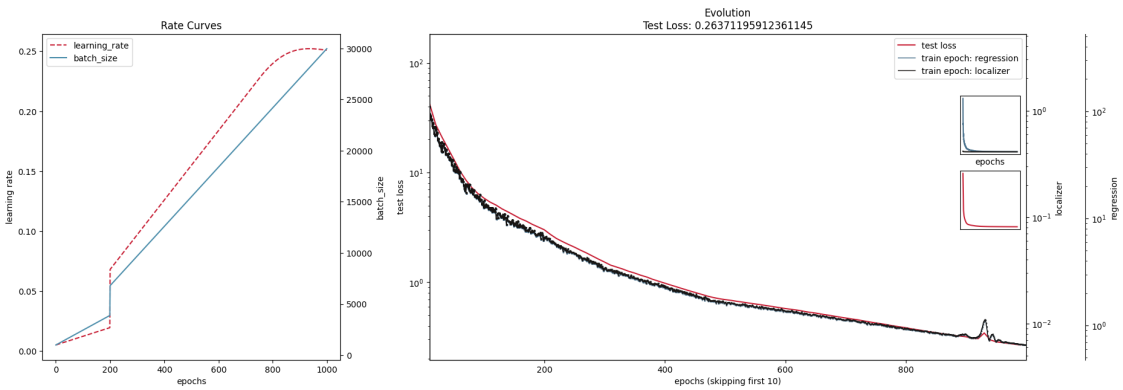


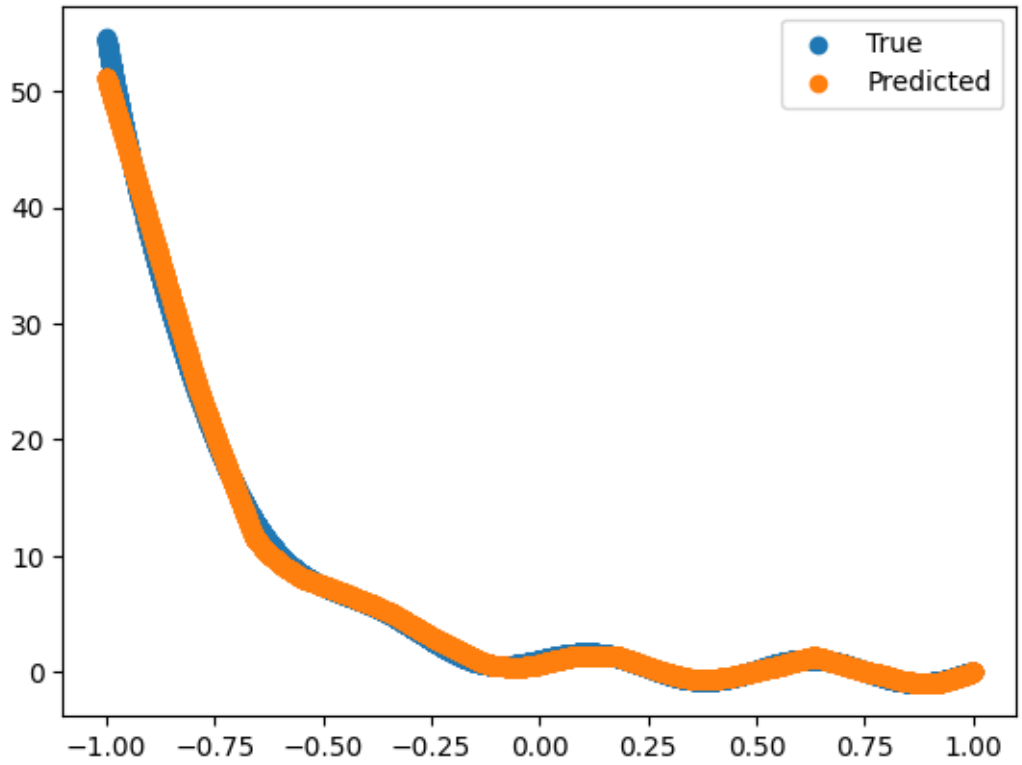


3.2 FL 2

[ ]:
```
act = torch.nn.ReLU()
delta = np.array([1]*len(fl_sizes))*0.00005
k = [ks[2]]*len(fl_sizes)
net = FL(device, fl_sizes, delta, k, opttype=opt, act = act, bias=True) #␣
 ↪expect 1.6% error rate
report = D.train(net, train_X, train_y, test_X, test_y, start_batch_size=1000)
ecran(net, test_X, test_y, report, classification=False)
plt.scatter(test_X.detach().cpu().numpy(),test_y.detach().cpu().
 ↪numpy(),label='True')
plt.scatter(test_X.detach().cpu().numpy(),net(test_X).detach().cpu().
 ↪numpy(),label='Predicted')
plt.legend()
plt.show()
```

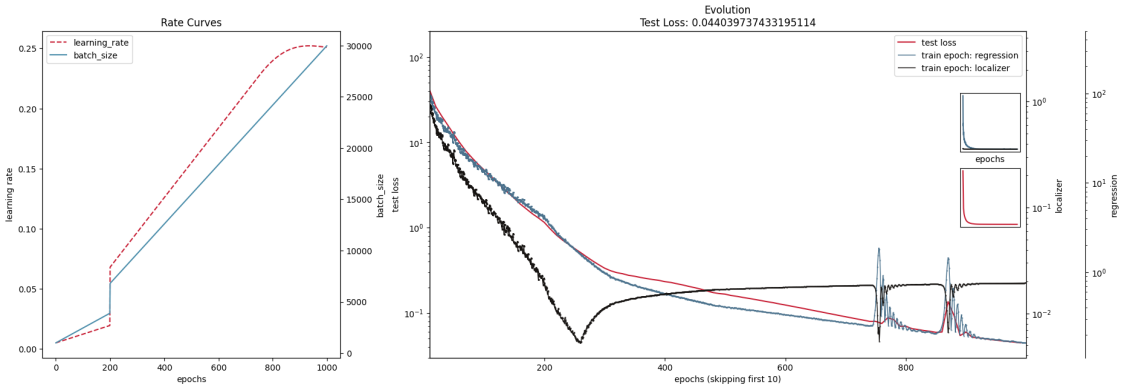
```
100%|      | 1000/1000 [00:46<00:00, 21.69it/s]
```

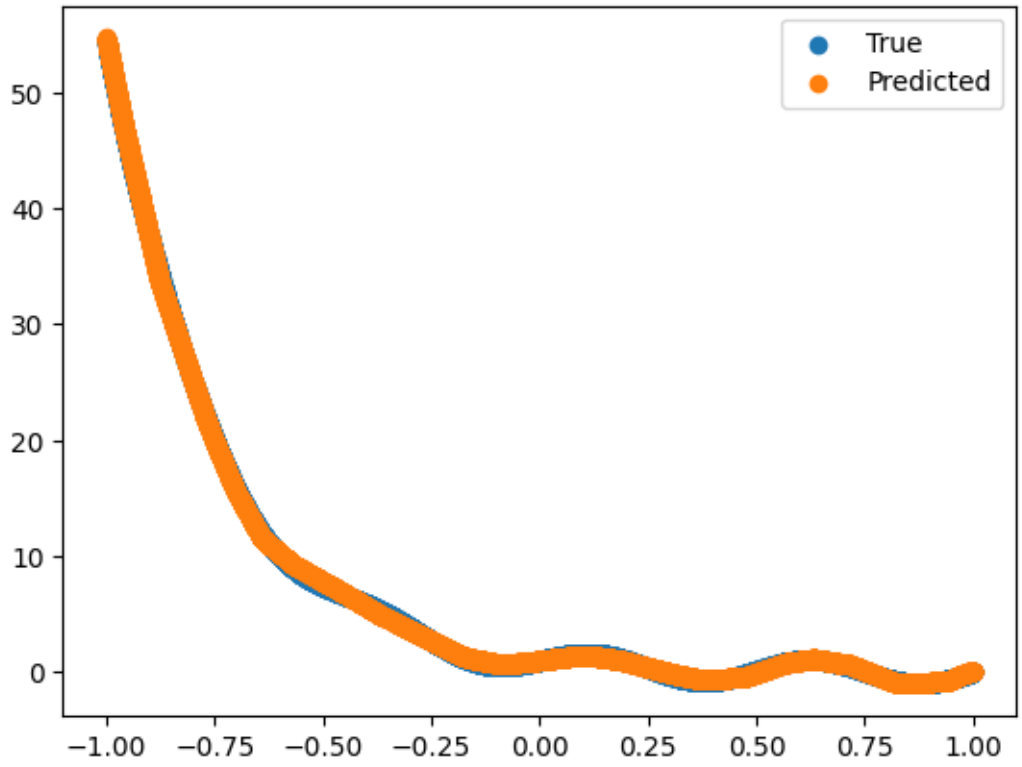




[ ]:
```
act = torch.nn.ReLU()
delta = np.array([1]*len(fl_sizes))*0.0005
k = [ks[2]]*len(fl_sizes)
net = FL(device, fl_sizes, delta, k, opttype=opt, act = act, bias=True) #␣
 ↪expect 1.6% error rate
report = D.train(net, train_X, train_y, test_X, test_y, start_batch_size=1000)
ecran(net, test_X, test_y, report, classification=False)
plt.scatter(test_X.detach().cpu().numpy(),test_y.detach().cpu().
 ↪numpy(),label='True')
plt.scatter(test_X.detach().cpu().numpy(),net(test_X).detach().cpu().
 ↪numpy(),label='Predicted')
plt.legend()
plt.show()
```

```
100%|      | 1000/1000 [00:45<00:00, 22.21it/s]
```

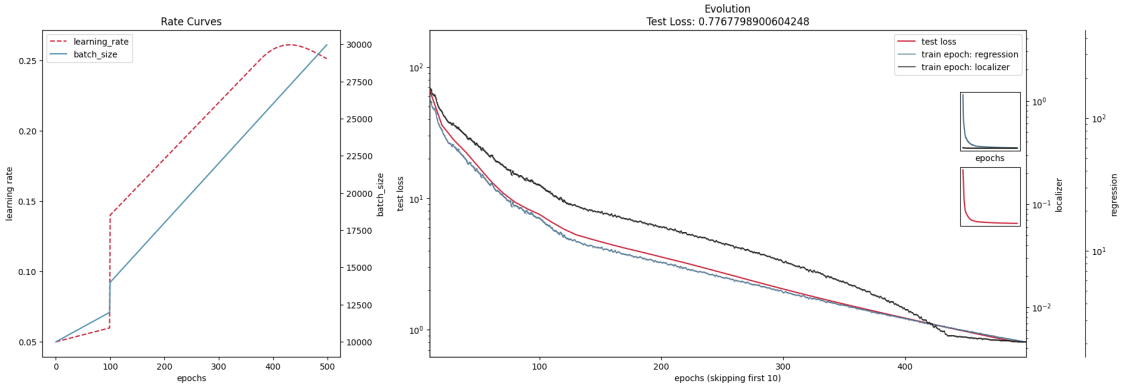


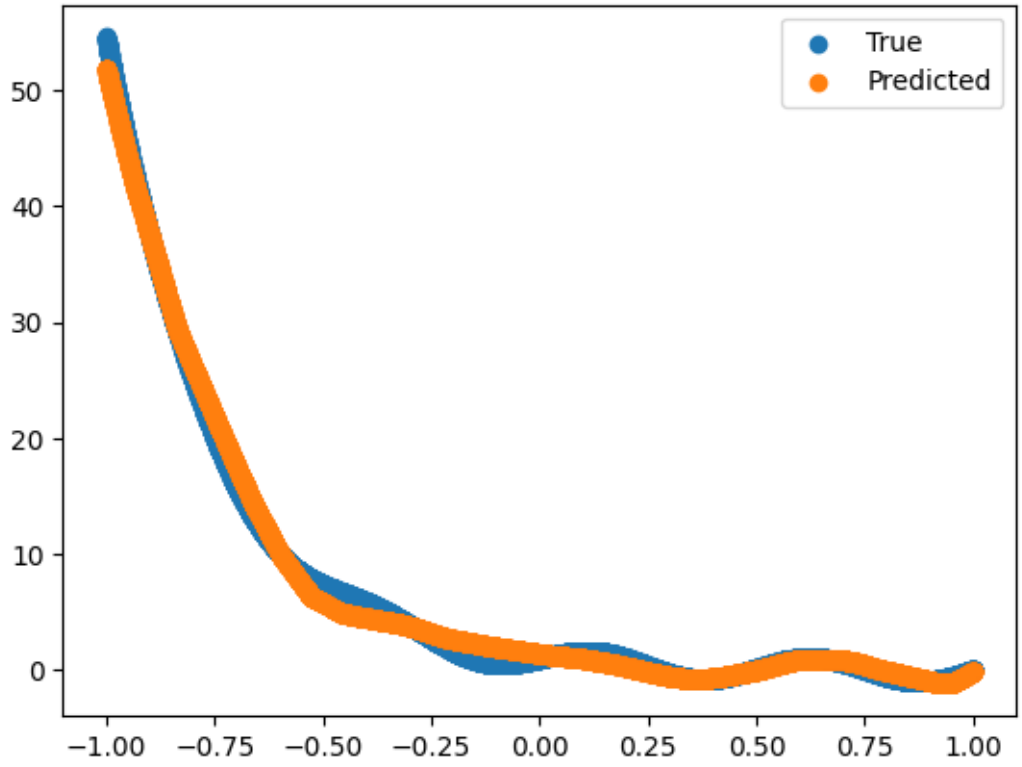


[ ]:
```
act = torch.nn.ReLU()
delta = np.array([1]*len(fl_sizes))*0.0005
k = [ks[2]]*len(fl_sizes)
net = FL(device, fl_sizes, delta, k, opttype=opt, act = act, bias=True) #␣
 ↪expect 1.6% error rate
D.epochs = 500
report = D.train(net, train_X, train_y, test_X, test_y, start_batch_size=10000)
D.epochs = 1000
```

```
ecran(net, test_X, test_y, report, classification=False)
plt.scatter(test_X.detach().cpu().numpy(),test_y.detach().cpu().
 ↪numpy(),label='True')
plt.scatter(test_X.detach().cpu().numpy(),net(test_X).detach().cpu().
 ↪numpy(),label='Predicted')
plt.legend()
plt.show()
```

```
100%|      | 500/500 [00:14<00:00, 33.68it/s]
```



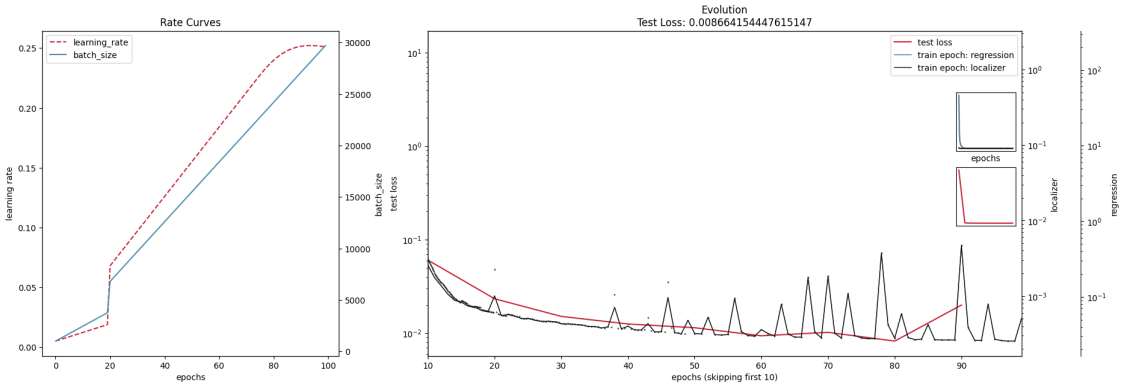


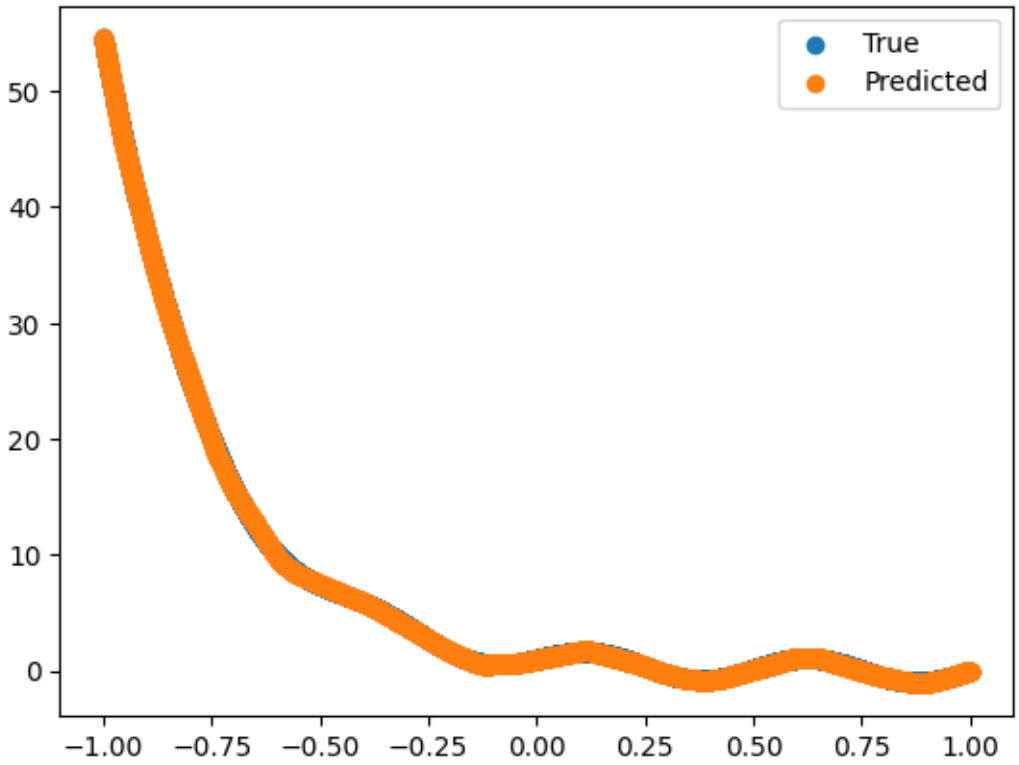
[ ]:

3.3 FL 2 Greedy

[ ]:
```
act = torch.nn.ReLU()
delta = np.array([1]*len(fl_sizes))*0.00005
k = [ks[1]]*len(fl_sizes)
net = FL(device, fl_sizes, delta, k, opttype=opt, act = act, bias=True) #␣
 ↪expect 1.6% error rate
D.epochs = 100
report = D.train(net, train_X, train_y, test_X, test_y, start_batch_size=1000,␣
 ↪repeat_epochs=40)
D.epochs = 1000
ecran(net, test_X, test_y, report, classification=False)
plt.scatter(test_X.detach().cpu().numpy(),test_y.detach().cpu().
 ↪numpy(),label='True')
plt.scatter(test_X.detach().cpu().numpy(),net(test_X).detach().cpu().
 ↪numpy(),label='Predicted')
plt.legend()
plt.show()
```

```
100%|      | 100/100 [02:58<00:00,  1.78s/it]
```



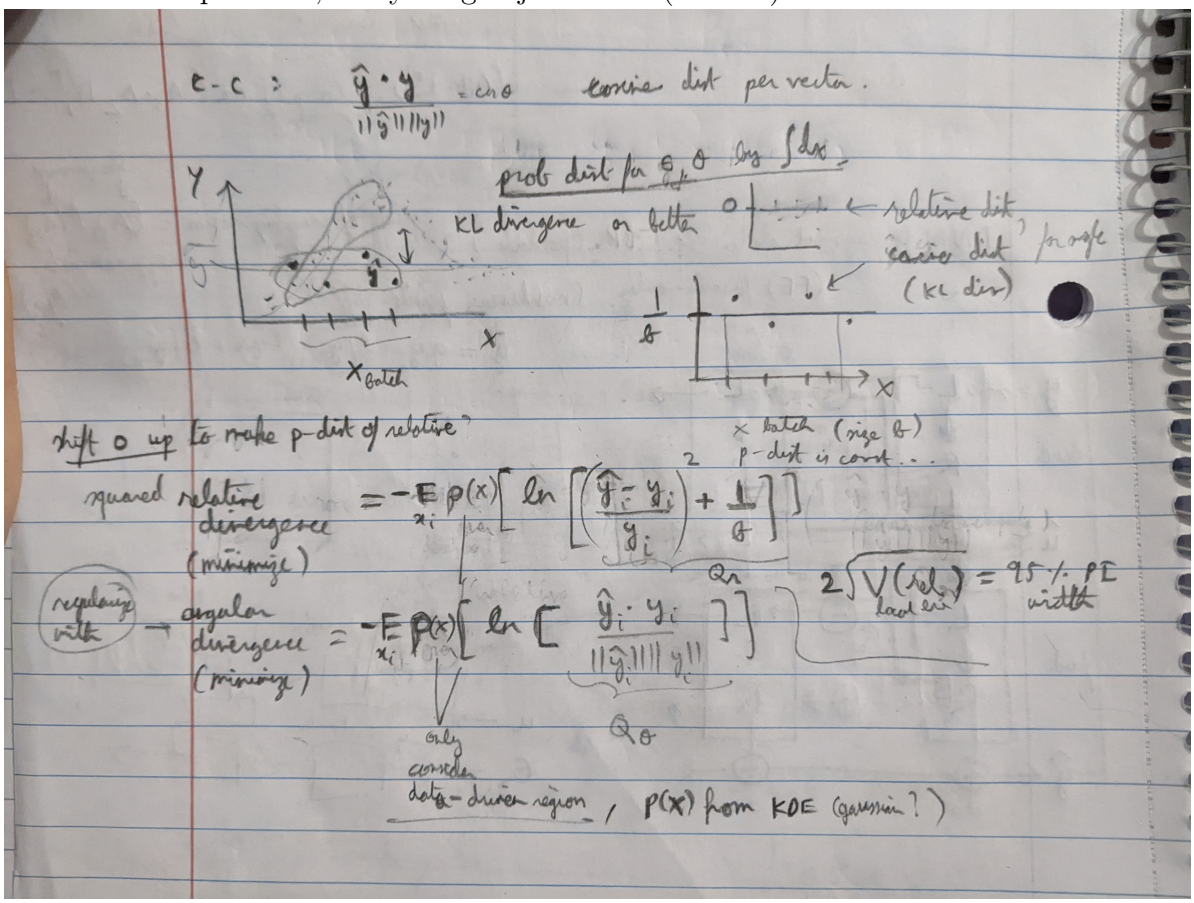

True
Predicted
50
40
30
20
10
0
−1.00
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75
1.00

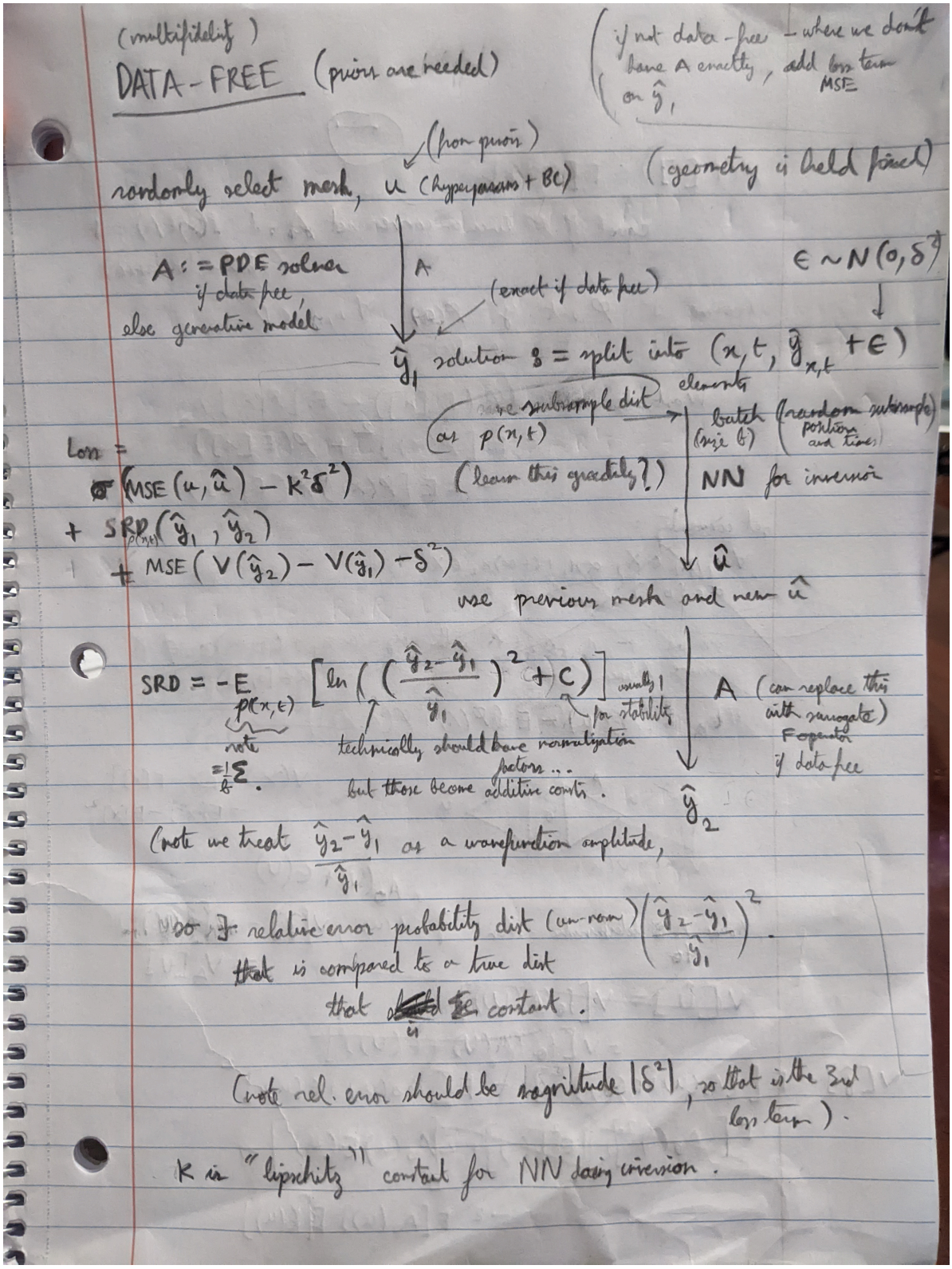
[ ]:

[ ]:

3. Forward Decomposed PDE-Inverter (Data-Free)

TODO : from this point on, everything is just notes (rewrite)!





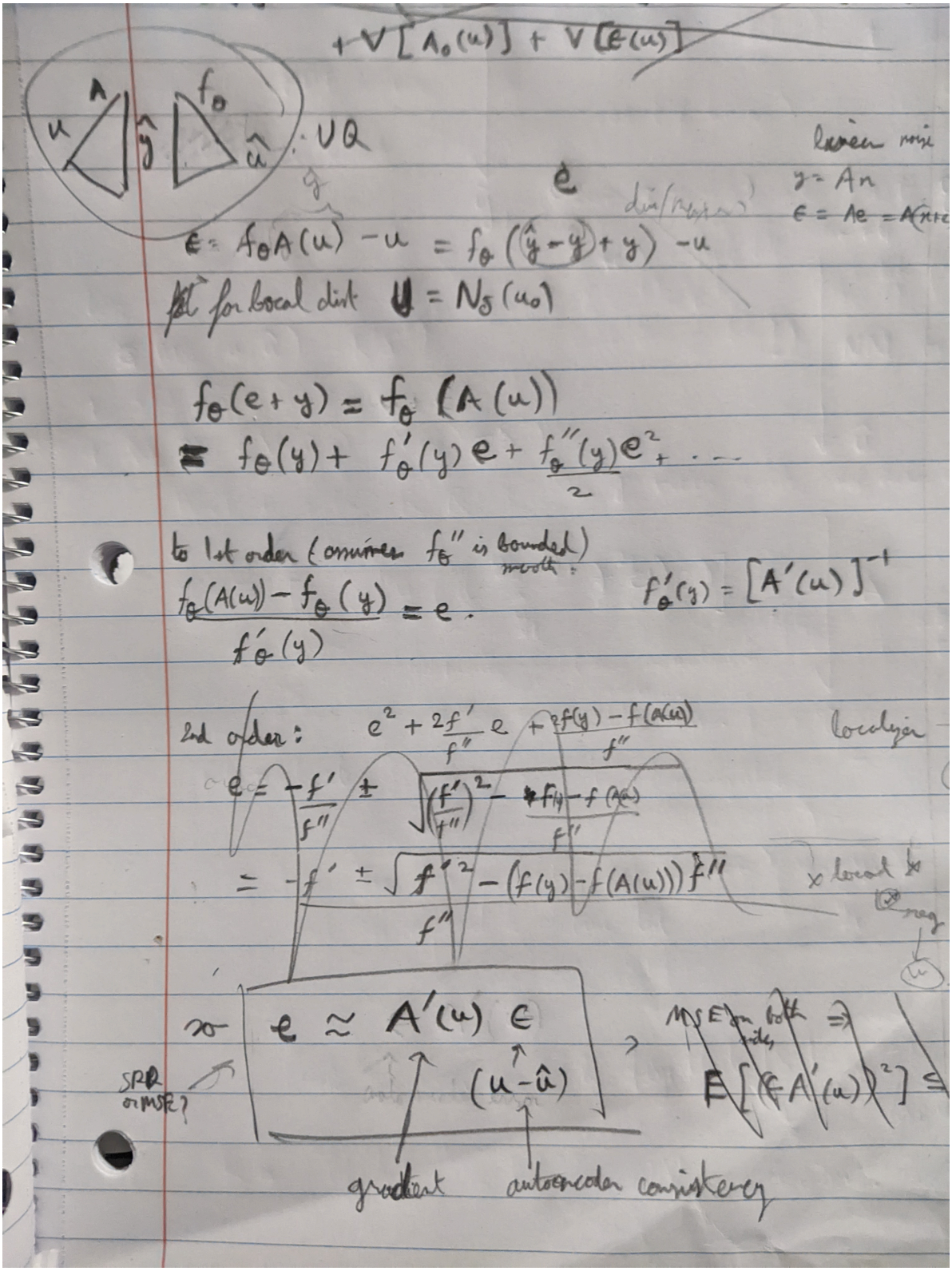
- Use FD to learn a T-Net (autoencoder), with a generative FD variant for the forward pass. - Goal is low-data semi/un-supervised learning, with OOD-based data augmentation. - Test on wafer scratch/defect detection? - Use a good noise model in forward pass to quantify uncertainty. (do a probability analysis) - Use ELBO loss and possibly

wavelet encoding from Paris Perdakis' work to make it resolution-independent. - Use fractal theory to learn fracture patterns in materials.

4. Forward Decomposed PDE-GAE (generative autoencoder)

- Now move to semi-sup PDE learning

5. PDE-UQ


+ V[A₀(u)] + V[ε(u)]
A
f_θ
u
ŷ
û
UQ
g
e
linear noise
y = An
ε = Ae = A(x+e
dim(range)
ε = f_θ A(u) − u = f_θ((ŷ − y) + y) − u
for local dist U = N_δ(u₀)
f_θ(e + y) ≈ f_θ(A(u))
≈ f_θ(y) + f_θ'(y) e + f_θ''(y) e²/2 + ...
to 1st order (assumes f_θ'' is bounded smooth)
(f_θ(A(u)) − f_θ(y)) / f_θ'(y) ≈ e.
f_θ'(y) = [A'(u)]⁻¹
2nd order:
e² + 2f'/f'' e + 2(f(y) − f(A(u)))/f''
e ≈ −f'/f'' ± √((f'/f'')² − 2(f(y) − f(A(u)))/f'')
≈ −f' ± √(f'² − (f(y) − f(A(u)))f'') / f''
localize
× local
neg
u
e ≈ A'(u) ε
(u − û)
MSE on both sides ⇒
E[(A'(u))²] ≤
SPR or MSE?
gradient
autoencoder consistency

6. Appendix

6.1. PAST DATA WITH ERRORS!

6.1.1. data 041523 Known ERRORS:

- Localizer loss had $k\delta$ instead of $k^2\delta^2$.

files affected:

- compare.ipynb
- fl greedy, fl sine, fl stability.ipynb
- fl, fl2, fl3, fl4, fl5.ipynb

PDF:./FDA/memo/plots/data$_0$41523/*compare.pdf*

7. Acknowledgements

Inspired by Dr. Hinton's paper [**?**]. This work follows from a discussion with Mr. C G Krishnanunni and Mr. Wesley Lao.

8. References