train_TangentNets

April 30, 2022

[1]:
```python
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import torch
import torch.nn as nn
from torch.optim import Adam
from torch.utils.data import DataLoader

from binarypredictor import split_functions
from binarypredictor.dataset import FunctionPairDataset
from binarypredictor.net import DerivativeNet, TangentNet
```

[2]:
```python
@torch.enable_grad()
def epoch(net, train_loader, loss_func, optimizer, f_func, g_func):
    """
    Training epoch of the network

    Parameters
    ----------
    net : TangentNet
        neural network to train
    train_loader : DataLoader
        training data
    loss_func : torch.nn loss function
        loss function
    optimizer : torch.optim optimizer
        optimizer
    f_func : callable
        function which is evaluated with the network outputs and compared to␣
↪g_func
    g_func : callable
        function which is evaluated at x and compared to f_func

    Returns
    -------
    float :
        mean epoch loss
```

```
    """

    epoch_losses = np.zeros([len(train_loader), ])

    for i, d in enumerate(train_loader):
        inp = torch.hstack((d[0][:, :, 0], d[0][:, :, 1]))  # network input
        out = net(inp)  # network output
        out = torch.clamp(out, min=1e-10, max=1.-1e-4)  # clamp outputs for␣
↪numerical stability

        # Evaluate the functions for the loss (common tangent equations)
        f = f_func(out, d[1][0])/d[2].unsqueeze(-1)
        g = g_func(x, d[1][1])/d[2].unsqueeze(-1)

        # Calculate the loss
        loss = loss_func(f, g)
        epoch_losses[i] = loss

        # Backward step
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

    return epoch_losses.mean()
```

[406]:
```
@torch.enable_grad()
def train(net, train_loader, test_loader, f_func, g_func, nr_epochs, lr,␣
↪print_every=10, net_filename='net_1.pth'):
    """
    Training of the network

    Parameters
    ----------
    net : TangentNet
        neural network to train
    train_loader : DataLoader
        training data
    test_loader : DataLoader
        test data
    f_func : callable
        function which is evaluated with the network outputs and compared to␣
↪g_func
    g_func : callable
        function which is evaluated at x and compared to f_func
    nr_epochs : int
        number of epochs to train
    lr : float
```

```python
        learning rate
    print_every : int
        multiple of epochs where losses are printed
    net_filename : str
        filename to save to net at

    Returns
    -------
    DerivativeNet :
        net with best training loss
    """

    loss_func = nn.L1Loss()
    optimizer = Adam(net.parameters(), lr=lr)

    losses = np.zeros([nr_epochs, ])
    test_losses = np.zeros([nr_epochs // print_every, ])

    best_loss = epoch(net, train_loader, loss_func, optimizer, f_func, g_func)
    best_net = net

    for i in range(nr_epochs):
        losses[i] = epoch(net, train_loader, loss_func, optimizer, f_func,␣
↪g_func)
        if losses[i] < best_loss:
            best_net = net
            best_loss = losses[i]

        if i % print_every == 0:
            print('Train loss : ', losses[i])
            test_losses[i // print_every] = test(best_net, test_loader, f_func,␣
↪g_func, loss_func)
            print('Test loss: ', test_losses[i // print_every])

            torch.save(best_net, net_filename)

    return best_net, losses, test_losses

@torch.no_grad()
def test(net, test_loader, f_func, g_func, metric):
    """
    Training of the network

    Parameters
    ----------
    net : TangentNet
        neural network to test
```

```
    test_loader : DataLoader
        test data
    f_func : callable
        function which is evaluated with the network outputs and compared to␣
↪g_func
    g_func : callable
        function which is evaluated at x and compared to f_func
    metric : callable
        metric to evaluate the network with

    Returns
    -------
    float :
        loss on the test set
    """

    test_losses = np.zeros([len(train_loader), ])

    for i, d in enumerate(test_loader):
        inp = torch.hstack((d[0][:, :, 0], d[0][:, :, 1]))  # network input
        out = net(inp)  # network output

        # Evaluate the functions for the loss (common tangent equations)
        f = f_func(out, d[1][0])/d[2].unsqueeze(-1)
        g = g_func(x, d[1][1])/d[2].unsqueeze(-1)

        # Calculate the loss
        loss = metric(f, g)
        test_losses[i] = loss

    return test_losses.mean()
```

[708]:
```
@torch.no_grad()
def predict(net_1, net_2, f, g, df, dg, scale=1., plot=False, acc=4,␣
↪threshold=0.3, k=15):
    """
    Predicts the equilibrium compositions of a binary system

    Parameters
    ----------
    net_1 : TangentNet
        network to predict equation 1
    net_2 : TangentNet
        network to predict equation 2
    f : callable
        function for f
    g : callable
```

```
        function for g
    df : torch.tensor
        first derivative values of f at x
    dg : torch.tensor
        first derivative values of g at x
    scale : float
        scaling factor so that the maximum function value is 1
    plot : bool
        whether to plot the results
    acc : int
        accuracy of output values (number of decimals)
    threshold : float
        threshold for the deviation from the tangent's slope and the functions' slopes
    k : int
        select topk results that meet the tangent condition in order to speed up the algorithm
    """
    net_1.eval(), net_2.eval()

    # Network input
    f_, g_ = f(x)/scale, g(x)/scale
    inp = torch.hstack((f_, g_))

    # Network outputs
    out_1 = net_1(inp)
    out_2 = net_2(inp)

    # Get the equilibrium compositions by calculating the points of intersections (by approximating as the intersection
    # of the lines connecting the values of out_1 and out_2 at sign changes)
    out_diff = out_1 - out_2
    idx = torch.where(abs(out_diff) < 0.1)[0][:-1]

    if len(idx) == 0:
        return torch.tensor([]), torch.tensor([])

    #x_f, x_g = (out_1[idx] + out_2[idx])/2, x[idx]
    x_f = torch.hstack((out_1[idx], out_2[idx], out_1[idx + 1], out_2[idx + 1],
                        (out_1[idx] + out_2[idx])/2, (out_1[idx + 1] + out_2[idx + 1])/2))
    x_g = torch.hstack((x[idx], x[idx], x[idx + 1], x[idx + 1], x[idx], x[idx + 1]))

    # Get the function values at the equilibria
    y_f, y_g = f(x_f)/scale, g(x_g)/scale
```

```
    # Get the slopes of the lines between the equilibria points
    slopes = (y_g - y_f)/(x_g - x_f)

    # Remove lines that are not tangents
    slope_cond = (abs(slopes - dg(x_g)/scale) <= threshold) & (abs(slopes - df(x_f)/scale) <= threshold)
    idx = torch.where(slope_cond)[0]

    # Recalculate x and y values for all points that are tangent points
    x_f, x_g = x_f[idx], x_g[idx]
    slope_dist = torch.sqrt((slopes[idx] - dg(x_g)/scale) ** 2 + (slopes[idx] - df(x_f)/scale) ** 2)

    # Only take the k best tangents to save time
    idx = torch.topk(slope_dist, min(k, len(slope_dist)), largest=False)[1]
    x_f, x_g = x_f[idx], x_g[idx]
    y_f, y_g = f(x_f)/scale, g(x_g)/scale
    slope_dist = torch.sqrt((slopes[idx] - dg(x_g)/scale) ** 2 + (slopes[idx] - df(x_f)/scale) ** 2)

    # Choose the best tangent if there are multiple results for the same tangent
    if len(x_f) > 0:
        x_eqs = torch.tensor(list(zip(x_f, x_g)))
        s_idx = torch.where(abs(torch.cdist(x_eqs, x_eqs)) < 0.05)

        left, right = s_idx[0], s_idx[1]
        left_unique = torch.unique(left)

        cis = []
        for i in left_unique:
            idx = torch.where(left == i)[0]
            add = right[idx]
            if len(add) > 0:
                cis.append(torch.argmin(slope_dist[add]))
            else:
                continue
            right = torch.tensor([r for r in right if r not in add])
            left = torch.tensor([l for l in left if l not in add])

        cis = torch.tensor(cis)

        x_f, x_g = torch.unique(x_f[cis]), torch.unique(x_g[cis])
        y_f, y_g = f(x_f)/scale, g(x_g)/scale

    # Plot the outputs
    if plot:
        plt.scatter(x, out_1.detach(), s=0.2)
```

```
        plt.scatter(x, out_2.detach(), s=0.2)
        plt.title('Network outputs')
        plt.xlabel('x\'\'')
        plt.ylabel('x\'')
        plt.grid()
        plt.show()

        plt.scatter(x, out_diff.detach(), s=0.2)
        plt.title('Difference between network outputs')
        plt.xlabel('x\'\'')
        plt.ylabel('Difference')
        plt.grid()
        plt.show()

        print('x\'_eq: ', np.round(x_f.tolist(), decimals=acc))
        print('x\'\'_eq: ', np.round(x_g.tolist(), decimals=acc))

        for x_f_eq, x_g_eq, y_f_eq, y_g_eq in zip(x_f, x_g, y_f, y_g):
            plt.plot([x_f_eq, x_g_eq], [y_f_eq, y_g_eq], 'ro-')
            plt.plot(x, f_)
            plt.plot(x, g_)
            plt.show()

    return x_f, x_g
```

[5]:
```
out_features = 500
in_features = out_features
```

[391]:
```
#fpd = FunctionPairDataset(n_functions=250000, filename="train_1m.csv",␣
 ↪overwrite=True, step=1/in_features)
#fpd.create_functions()

#fpd_test = FunctionPairDataset(n_functions=1000, filename="test_1m.csv",␣
 ↪overwrite=True, step=1/in_features)
#fpd_test.create_functions()
```

[407]:
```
train_loader = DataLoader(fpd, batch_size=1028)
test_loader = DataLoader(fpd_test, batch_size=1028)
net_1 = TangentNet(train=True, in_features=in_features * 2,␣
 ↪out_features=out_features, hidden_size_linear=500, hidden_layers=2)
net_2 = TangentNet(train=True, in_features=in_features * 2,␣
 ↪out_features=out_features, hidden_size_linear=500, hidden_layers=2)
```

[408]:
```
x = torch.arange(1e-10, 1., step=fpd_test.step)
```

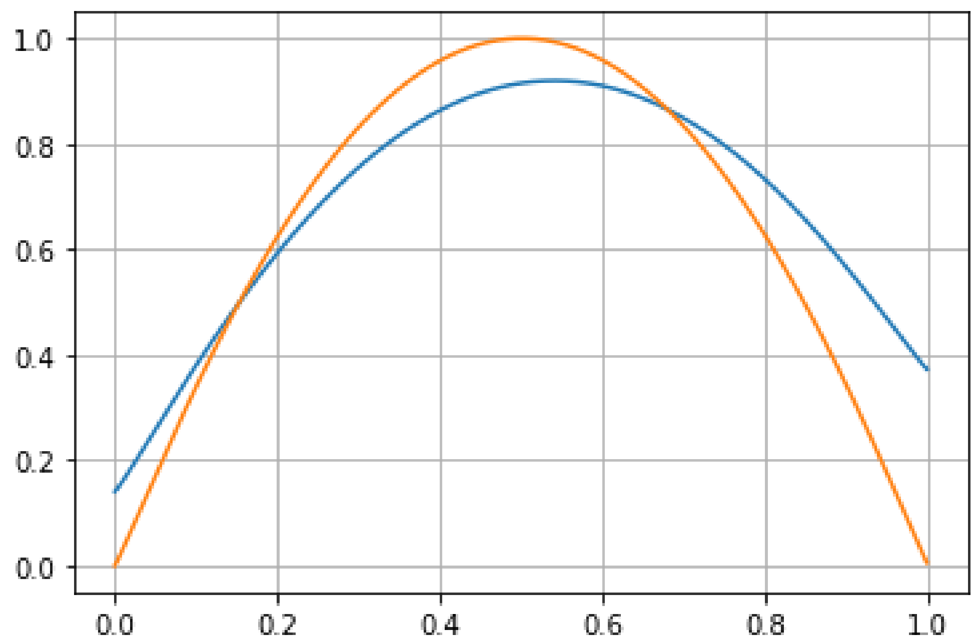
[ ]:
```
# Hyperparameters
nr_epochs = 250
```

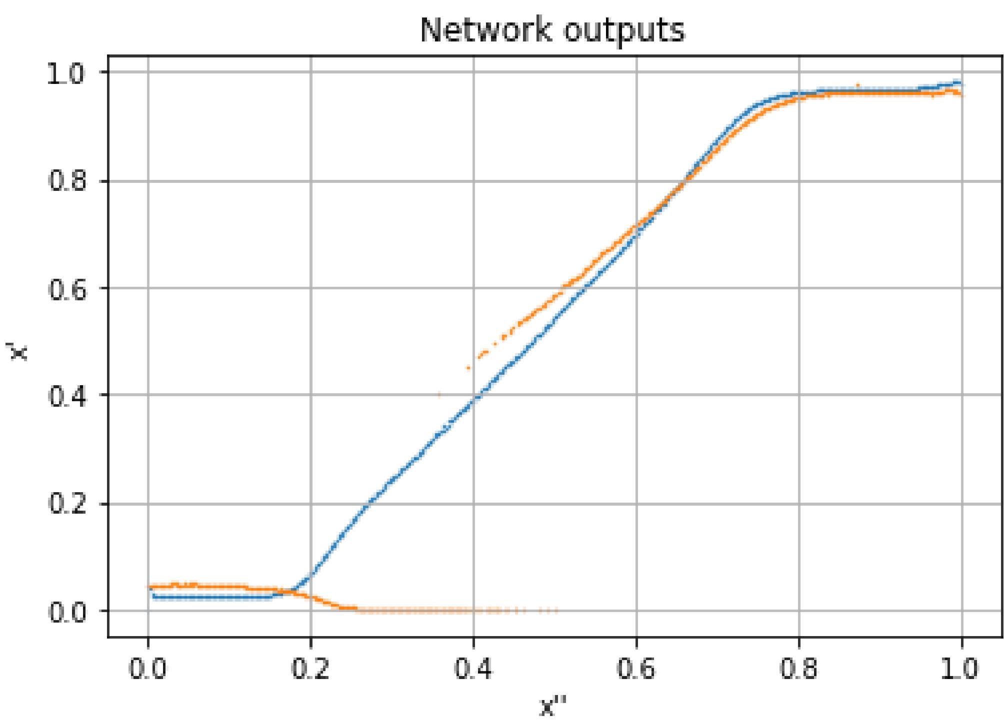
```
lr = 1e-3

# Train for equation 1
func_1 = lambda x_, d: fpd.first_derivative(**d, x=x_)
best_net_1, losses_1, test_losses_1 = train(net_1, train_loader, test_loader,␣
↪func_1, func_1, nr_epochs, lr, print_every=10, net_filename='net_1m.pth')

print('Trained network 1 \n')

# Train for equation 2
func_2 = lambda x_, d: fpd.base_function(**d, x=x_) - x_ * fpd.
↪first_derivative(**d, x=x_)
best_net_2, losses_2, test_losses_2 = train(net_2, train_loader, test_loader,␣
↪func_2, func_2, nr_epochs, lr, print_every=10, net_filename='net_2m.pth')

print('Trained network 2')
```

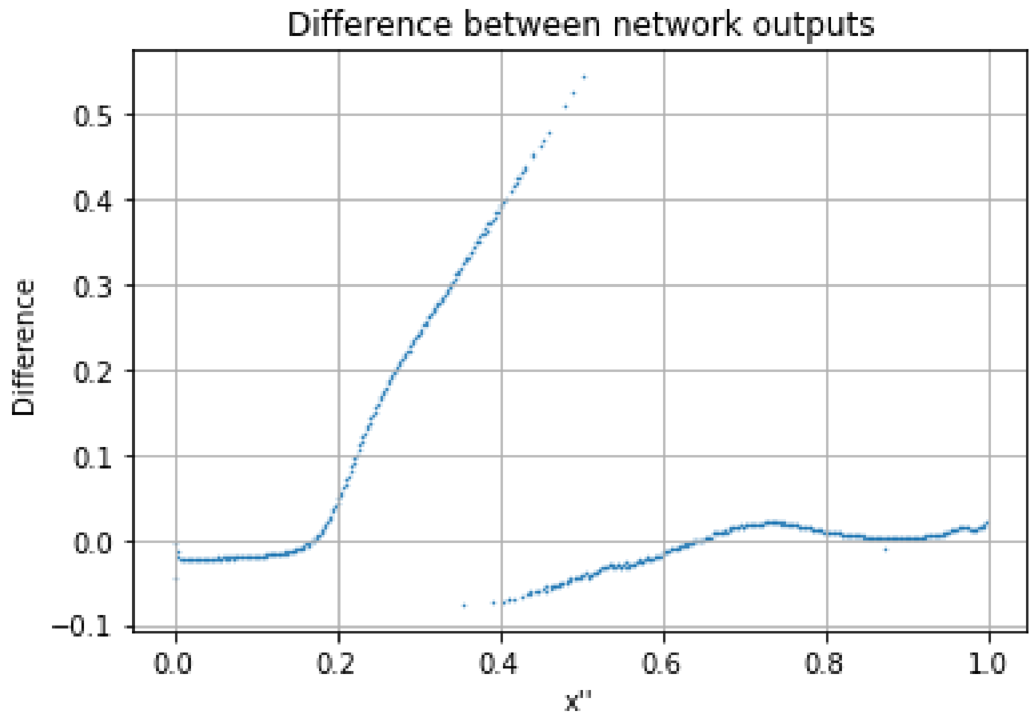
[410]:
```
#torch.save(losses_1, 'losses_1.txt')
#torch.save(test_losses_1, 'test_losses_1.txt')
#torch.save(losses_2, 'losses_2.txt')
#torch.save(test_losses_2, 'test_losses_2.txt')
```

[459]:
```
net_1 = torch.load('net_1.pth')
net_2 = torch.load('net_2.pth')
```

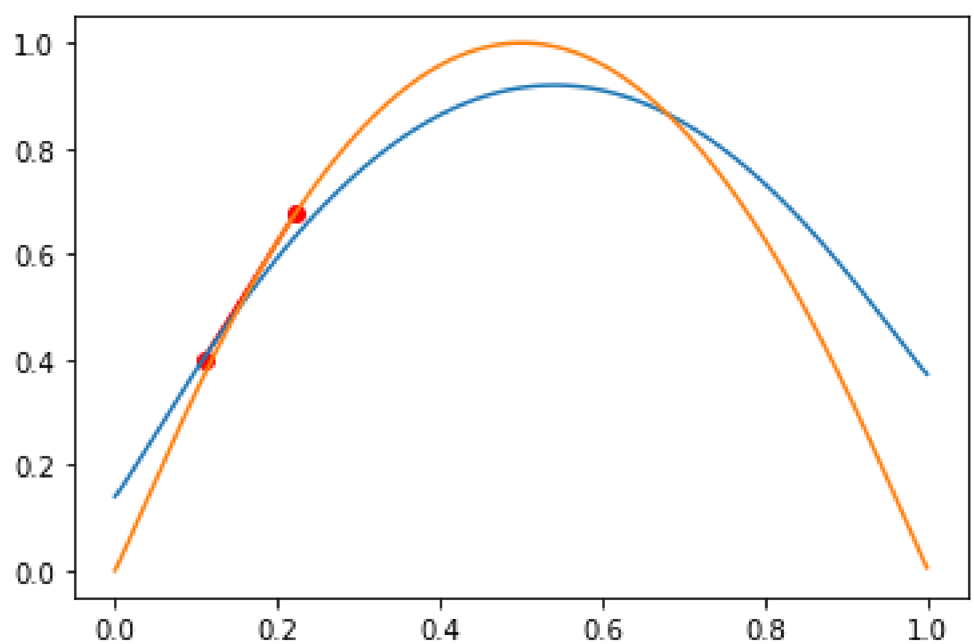
[709]:
```
data = fpd_test

d = data[30]
scale = d[2].unsqueeze(-1)

f = data.base_function(**d[1][0])/scale
g = data.base_function(**d[1][1])/scale
plt.plot(x, f)
plt.plot(x, g)
plt.grid()
plt.show()

f = lambda x_: data.base_function(**d[1][0], x=x_)/scale
g = lambda x_: data.base_function(**d[1][1], x=x_)/scale
df = lambda x_: data.first_derivative(**d[1][0], x=x_)/scale
dg = lambda x_: data.first_derivative(**d[1][1], x=x_)/scale

x_f, x_g = predict(net_1, net_2, f, g, df, dg, plot=True, threshold=.8, k=100)
```

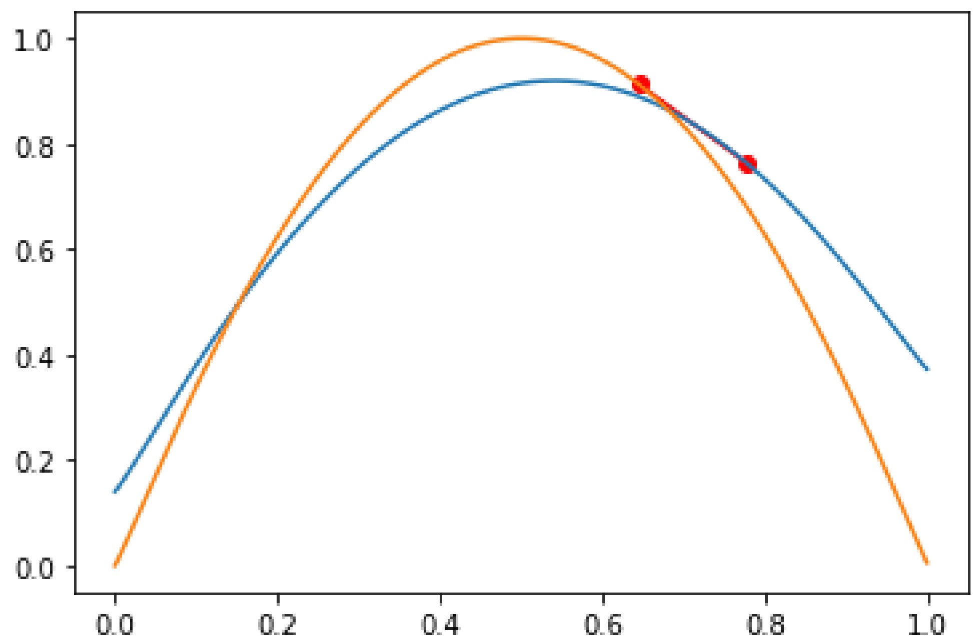

1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0


Network outputs
x'
x''
1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0




```
x'_eq:  [0.1091 0.7759]
x''_eq:  [0.222 0.644]
```





[16]:
```
import time
import timeit
```

[252]:
```
real_data = pd.read_csv('auag.txt', delimiter='\t')
```

[735]:
```
start_time = time.time()

x_fs = []
x_gs = []
ts = []

R = 8.3143

gf = lambda t: 3815.93 + 109.3029 * t - 1.044523e-20 * t ** 7 - (-7209.5 + 118.
 ↪2007 * t)
gg = lambda t: -3352 + 215.88 * t - 3.5899325e-21 * t ** 7 - (-15745 + 225.14 *␣
 ↪t)

t_range = real_data['LIQUID + FCC_A1']
for t in t_range:

    f = lambda x: (1 - x) * gf(t) + x * gg(t) + R * t * ((1 - x) * torch.log(1␣
 ↪- x) + x * torch.log(x)) + (1 - x) * x * (-16402 + 1.14 * t)
    g = lambda x: R * t * ((1 - x) * torch.log(1 - x) + x * torch.log(x)) + (1␣
 ↪- x) * x * (-15599)
```

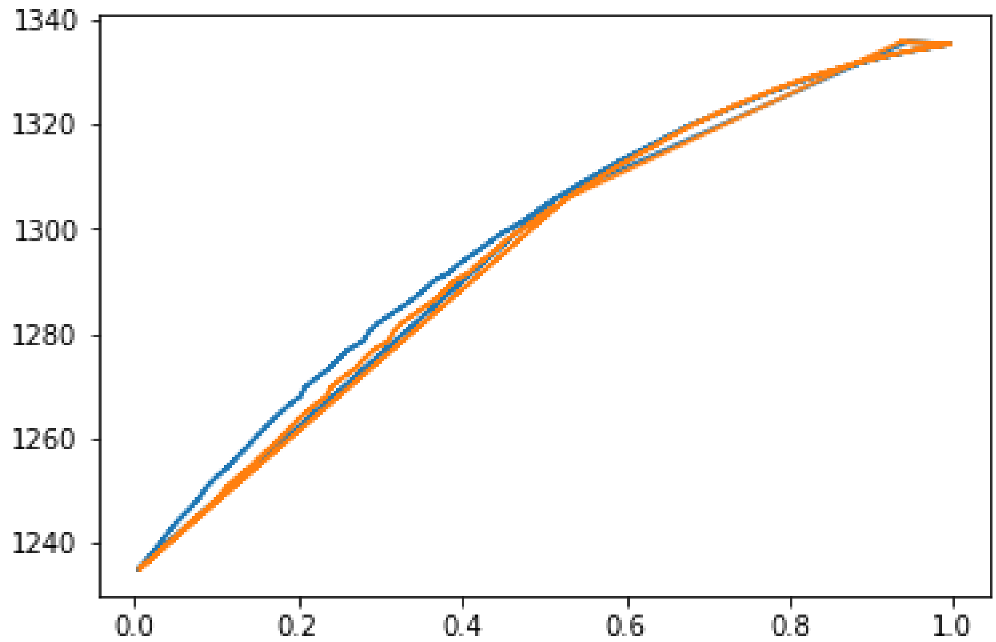
```python
    df = lambda x: -16402 - gf(t) + gg(t) + 1.14 * t + 32804 * x - 2.28 * t * x␣
↪- R * t * torch.log(1 - x) + R * t * torch.log(x)
    dg = lambda x: 15599 * (-1 + 2 * x) - R * t * torch.log(1 - x) + R * t *␣
↪torch.log(x)

    scale = max(torch.max(abs(f(x))), torch.max(abs(g(x))))

    x_f, x_g = predict(net_1, net_2, f, g, df, dg, scale=scale, plot=False,␣
↪threshold=0.3, k=1)

    x_fs.append(x_f) if len(x_f) > 0 else None
    x_gs.append(x_g) if len(x_g) > 0 else None
    ts.append(t) if len(x_f) > 0 else None

print(time.time() - start_time)
```

```
0.6649613380432129
```

[736]:
```python
x_fs_ = []
for x_ in x_fs:
    x_fs_.append(x_[0])

x_gs_ = []
for x_ in x_gs:
    x_gs_.append(x_[0])
```

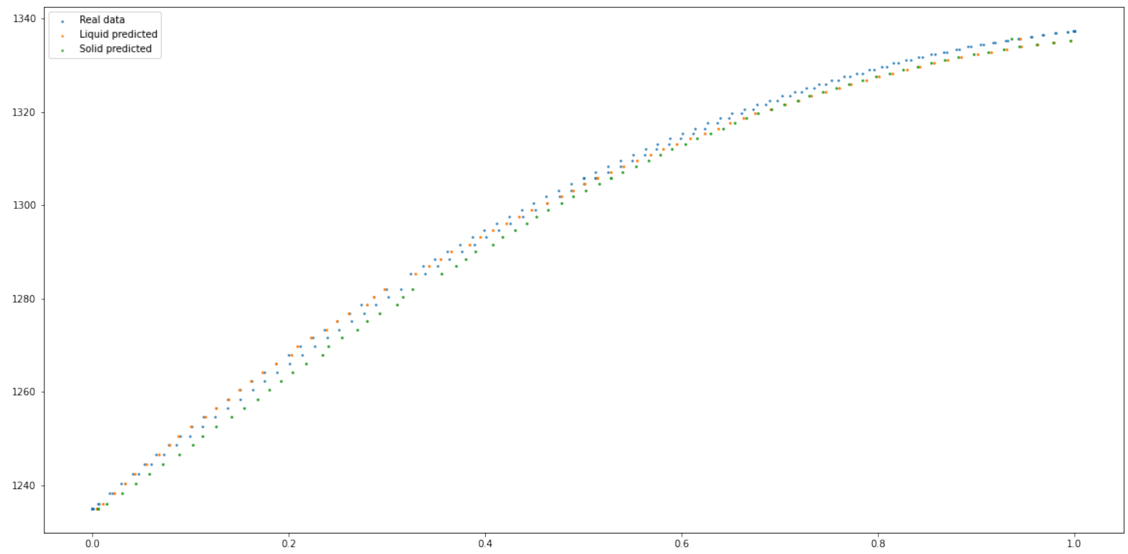
[737]:
```python
plt.plot(x_fs_, ts)
plt.plot(x_gs_, ts)
```

[737]:
```
[<matplotlib.lines.Line2D at 0x25a14146280>]
```




```
[738]: fig, ax = plt.subplots(1, 1, figsize=(20, 10))
       s = 2
       ax.scatter(real_data['Mole fraction Au'], real_data['LIQUID + FCC_A1'], s=s,␣
        ↪label='Real data')
       #ax.scatter(real_data['Mole fraction Au [1]'], real_data['Tieline'], s=s,␣
        ↪label='Real data')
       ax.scatter(x_fs_, ts, s=s, label='Liquid predicted')
       ax.scatter(x_gs_, ts, s=s, label='Solid predicted')
       ax.legend()
```

```
[738]: <matplotlib.legend.Legend at 0x25a16864d30>
```



[739]:

```
r_x = []

for x_, r_t in zip(real_data['Mole fraction Au'], real_data['LIQUID + FCC_A1']):
    if r_t in ts:
        r_x.append(x_)

x_fs_t = torch.tensor(x_fs_)
r_x_t = torch.tensor(r_x)

print('Mean error: ', nn.L1Loss()(x_fs_t, r_x_t).item())
print('Mean squared error: ', nn.MSELoss()(x_fs_t, r_x_t).item())
print('Max deviation: ' , torch.max(abs(x_fs_t - r_x_t)).item())
print('Min deviation: ' , torch.min(abs(x_fs_t - r_x_t)).item())
```

```
Mean error:  0.015526720322668552
Mean squared error:  0.00048736718599684536
Max deviation:  0.06612342596054077
Min deviation:  0.00018846988677978516
```