

pytorch - nn

Lecture 23

Dr. Colin Rundel

Odds & Ends

Torch models

Implementation details:

- Models are implemented as a class inheriting from `torch.nn.Module`
- Must implement constructor and `forward()` method
 - `__init__()` should call parent constructor via `super()`
 - Use `torch.nn.Parameter()` to indicate model parameters
 - `forward()` should implement the model - constants + parameters -> `return` predictions

Fitting procedure:

- For each iteration of solver:
 - Get current predictions via a call to `forward()` or equivalent.
 - Calculate a (scalar) loss or equivalent
 - Call `backward()` method on loss
 - Use built-in optimizer (`step()` and then `zero_grad()` if necessary)

From last time

```
1 class Model(torch.nn.Module):
2     def __init__(self, X, y, beta=None):
3         super().__init__()
4         self.X = X
5         self.y = y
6         if beta is None:
7             beta = torch.zeros(X.shape[1])
8             beta.requires_grad = True
9             self.beta = torch.nn.Parameter(beta)
10
11     def forward(self, X):
12         return X @ self.beta
13
14     def fit(self, opt, n=1000, loss_fn = torch.nn.MSELoss()):
15         losses = []
16         for i in range(n):
17             loss = loss_fn(
18                 self(self.X).squeeze(),
19                 self.y.squeeze()
20             )
21             loss.backward()
```

What is `self(X)`?

This is (mostly) just short hand for calling `self.forward(X)` to generate the output tensors from the current value(s) of the parameters.

This is done via the `__call__()` method in the `torch.nn.Module` class. `__call__()` allows python classes to be invoked like functions.

```
1 class greet:
2     def __init__(self, greeting):
3         self.greeting = greeting
4     def __call__(self, name):
5         return self.greeting + " " + name
```

```
1 hello = greet("Hello")
2 hello("Jane")
```

'Hello Jane'

```
1 gm = greet("Good morning")
2 gm("Bob")
```

'Good morning Bob'

MNIST & Logistic models

MNIST handwritten digits - simplified

```
1 from sklearn.datasets import load_digits
2 digits = load_digits()
```

```
1 X = digits.data
2 X.shape
```

(1797, 64)

```
1 X[0:2]
```

```
array([[ 0.,  0.,  5., 13.,  9.,  1.,  0.,
         0.,  0.,  0., 13., 15., 10., 15.,
         5.,  0.,  0.,  3., 15.,  2.,  0.,
        11.,  8.,  0.,  0.,  4., 12.,  0.,
         0.,  8.,  8.,  0.,  0.,  5.,  8.,
         0.,  0.,  9.,  8.,  0.,  0.,  4.,
        11.,  0.,  1., 12.,  7.,  0.,  0.,
         2., 14.,  5., 10., 12.,  0.,  0.,
         0.,  0.,  6., 13., 10.,  0.,  0.,
         0.],
       [ 0.,  0.,  0., 12., 13.,  5.,  0.,
         0.,  0.,  0.,  0., 11., 16.,  9.,
         0.,  0.,  0.,  0.,  3., 15., 16.,
         6.,  0.,  0.,  0.,  7., 15., 16.,
        16.,  2.,  0.,  0.,  0.,  0.,  1.,
        16., 16.,  3.,  0.,  0.,  0.,  0.,
         1., 16., 16.,  6.,  0.,  0.,  0.,
         0.,  1., 16., 16.,  6.,  0.,  0.]
```

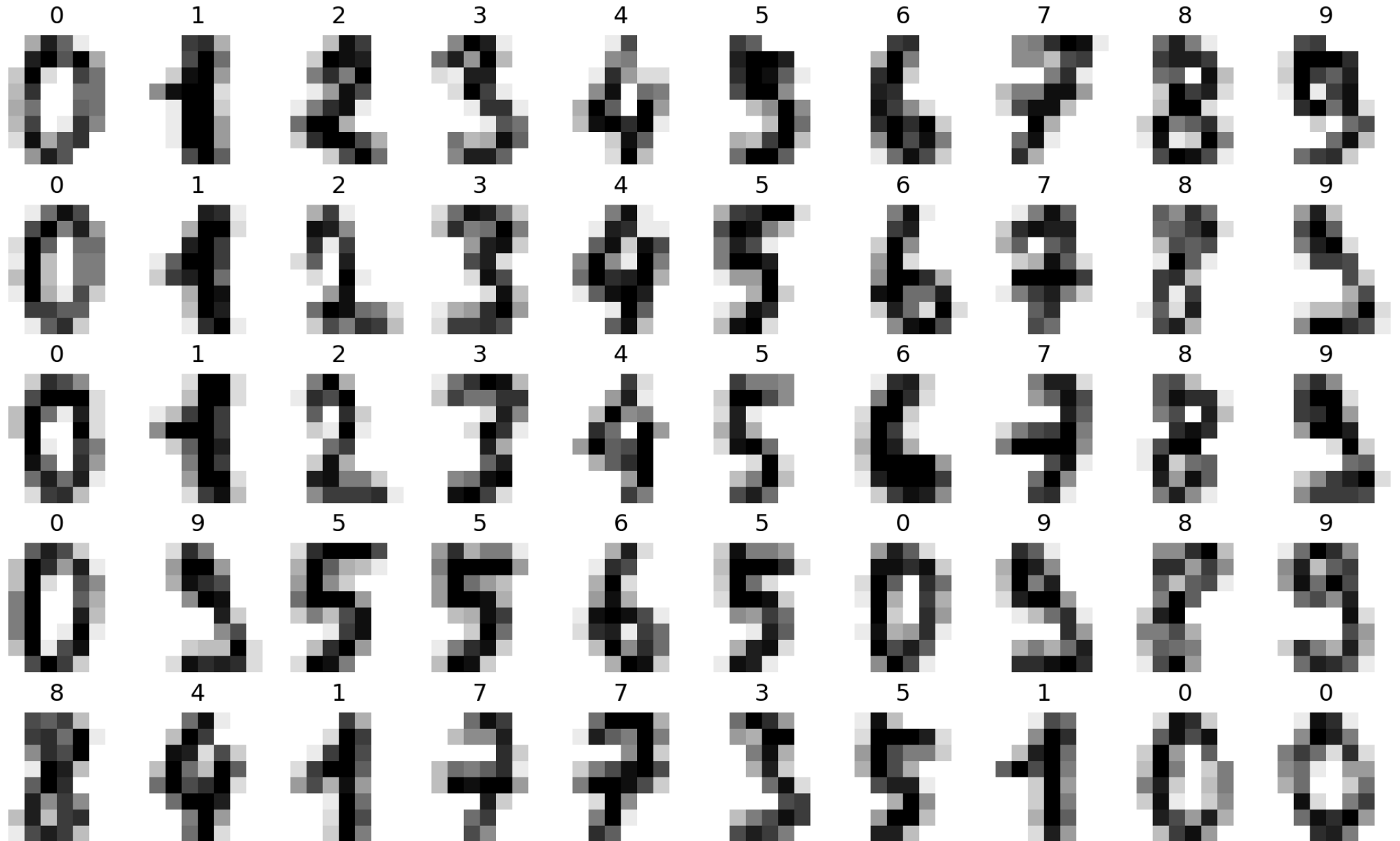
```
1 y = digits.target
2 y.shape
```

(1797,)

```
1 y[0:10]
```

```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

Example digits



Test train split

```
1 from sklearn.model_selection import train_test_split
2
3 X_train, X_test, y_train, y_test = train_test_split(
4     X, y, test_size=0.20, shuffle=True, random_state=1234
5 )
```

```
1 X_train.shape
```

(1437, 64)

```
1 y_train.shape
```

(1437,)

```
1 X_test.shape
```

(360, 64)

```
1 y_test.shape
```

(360,)

```
1 from sklearn.linear_model import LogisticRegression
2 from sklearn.metrics import accuracy_score
```

```
1 lr = LogisticRegression(
2     penalty=None
3 ).fit(
4     X_train, y_train
5 )
```

```
1 accuracy_score(y_train, lr.predict(X_train))
```

1.0

```
1 accuracy_score(y_test, lr.predict(X_test))
```

0.9583333333333334

As Torch tensors

```
1 X_train = torch.from_numpy(X_train).float()  
2 y_train = torch.from_numpy(y_train)  
3 X_test = torch.from_numpy(X_test).float()  
4 y_test = torch.from_numpy(y_test)
```

```
1 X_train.shape
```

```
torch.Size([1437, 64])
```

```
1 y_train.shape
```

```
torch.Size([1437])
```

```
1 X_test.shape
```

```
torch.Size([360, 64])
```

```
1 y_test.shape
```

```
torch.Size([360])
```

```
1 X_train.dtype
```

```
torch.float32
```

```
1 y_train.dtype
```

```
torch.int64
```

```
1 X_test.dtype
```

```
torch.float32
```

```
1 y_test.dtype
```

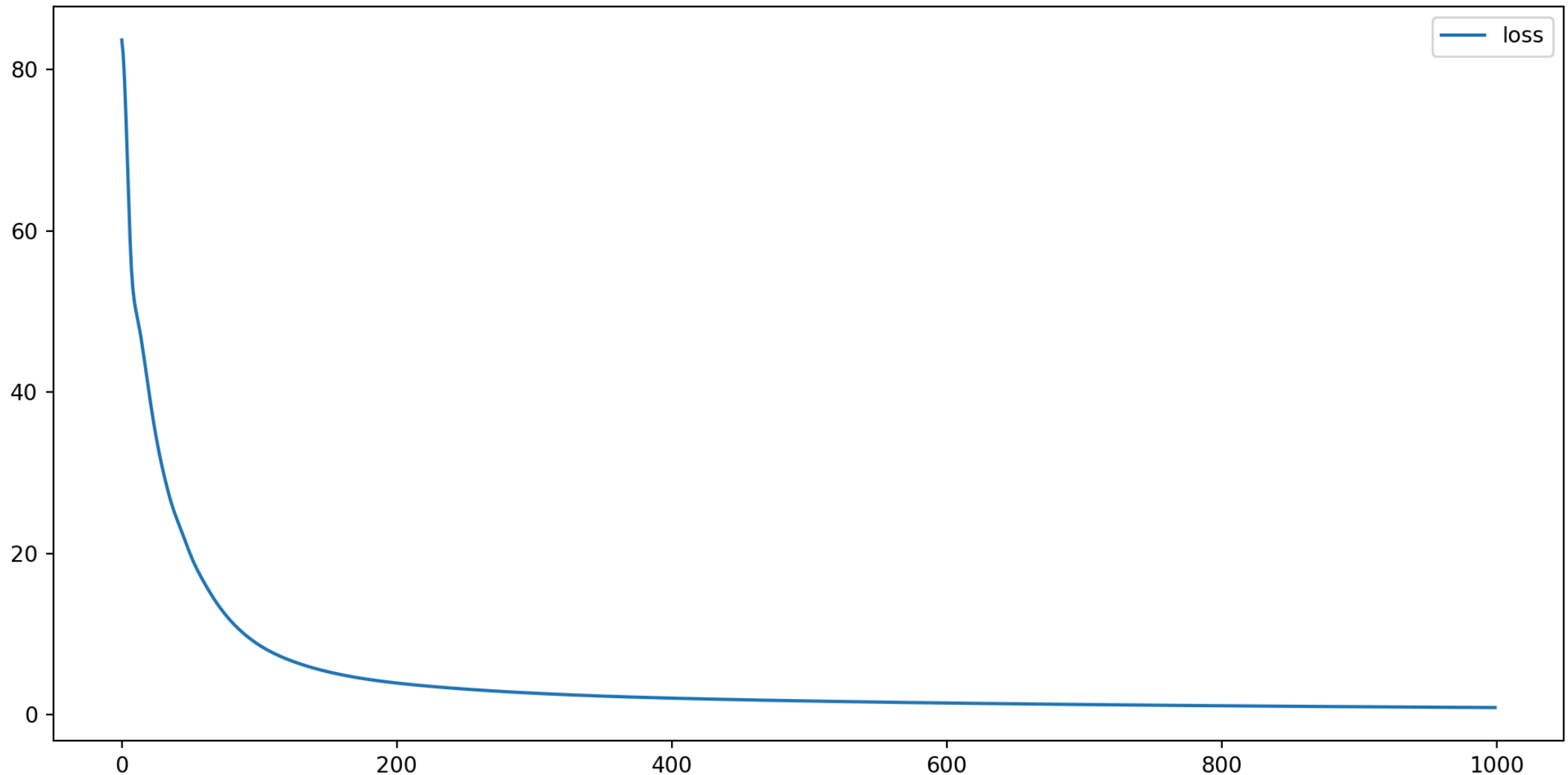
```
torch.int64
```

PyTorch Model

```
1 class mnist_model(torch.nn.Module):
2     def __init__(self, input_dim, output_dim):
3         super().__init__()
4         self.beta = torch.nn.Parameter(
5             torch.randn(input_dim, output_dim, requires_grad=True)
6         )
7         self.intercept = torch.nn.Parameter(
8             torch.randn(output_dim, requires_grad=True)
9         )
10
11     def forward(self, X):
12         return (X @ self.beta + self.intercept).squeeze()
13
14     def fit(self, X_train, y_train, X_test, y_test, lr=0.001, n=1000):
15         opt = torch.optim.SGD(self.parameters(), lr=lr, momentum=0.9)
16         losses = []
17
18         for i in range(n):
19             opt.zero_grad()
20             loss = torch.nn.CrossEntropyLoss()(self(X_train), y_train)
21             loss.backward()
```

Cross entropy loss

```
1 model = mnist_model(64, 10)
2 l = model.fit(X_train, y_train, X_test, y_test)
```



Cross entropy loss

From the pytorch documentation:

$$\ell(x, y) = L = \{l_1, \dots, l_N\}^\top, \quad l_n = -w_{y_n} \log \frac{\exp(x_{n,y_n})}{\sum_{c=1}^C \exp(x_{n,c})}$$

$$\ell(x, y) = \begin{cases} \sum_{n=1}^N \frac{1}{\sum_{n=1}^N w_{y_n} \cdot 1\{y_n \neq \text{ignore_index}\}} l_n, & \text{if reduction = 'mean'} \\ \sum_{n=1}^N l_n, & \text{if reduction = 'sum'} \end{cases}$$

Out-of-sample accuracy

```
1 model(X_test)
```

```
tensor([[ -1.0983e+01, -2.0008e+01,
          -1.6966e+01, -1.1108e+01,
          -4.7596e+01, -3.2255e+01,
          -3.8194e+01,  3.8499e+01,
          -2.3000e+01, -3.7694e+01],
        [-6.8701e+01,  2.7801e+01,
          -9.0201e+01, -1.3878e+01,
          -1.9980e+01, -2.5793e+01,
          -7.5506e+01, -4.5115e+01,
           1.8130e+01,  5.1021e+01],
        [-2.9823e+01, -3.8137e+01,
          -1.9869e+01, -2.1973e+01,
          -4.1130e+01, -6.2912e+01,
          -9.2013e+01,  4.0552e+01,
          -3.6866e+01, -2.5443e+01],
        [ 1.0965e+01, -9.5212e+00,
           5.1235e+00,  1.9886e+00,
           2.0020e+01,  0.1600e+00])
```

```
1 val, index = torch.max(model(X_test), dim=1)
2 index
```

```
tensor([7, 9, 7, 6, 0, 2, 4, 3, 6, 3, 7, 8, 7,
         9, 4, 3, 1, 7, 8, 4, 0, 3, 9, 1, 3, 6,
         6, 0, 5, 4, 1, 2, 1, 2, 3, 2, 7, 6, 1,
         8, 6, 4, 4, 0, 9, 1, 8, 5, 4, 4, 4, 1,
         7, 6, 8, 2, 9, 9, 7, 0, 8, 3, 1, 8, 8,
         1, 3, 9, 9, 3, 9, 6, 9, 5, 8, 1, 9, 2,
         1, 3, 8, 7, 3, 3, 1, 7, 7, 5, 8, 2, 6,
         1, 9, 1, 6, 4, 5, 2, 2, 4, 5, 6, 4, 6,
         5, 9, 2, 4, 1, 0, 7, 6, 1, 2, 9, 5, 2,
         5, 0, 3, 2, 7, 6, 4, 3, 2, 1, 1, 6, 7,
         6, 8, 1, 4, 7, 5, 0, 9, 1, 0, 5, 6, 7,
         6, 3, 8, 3, 2, 0, 4, 0, 1, 5, 4, 6, 1,
         1, 1, 6, 1, 7, 0, 0, 7, 9, 5, 4, 1, 3,
         8, 6, 4, 9, 1, 5, 7, 4, 7, 4, 0, 2, 2,
         1, 1, 4, 6, 3, 5, 5, 9, 4, 5, 5, 9, 3,
         9, 2, 1, 2, 0, 8, 2, 8, 6, 2, 4, 6, 8,
         3, 9, 1, 0, 8, 1, 8, 5, 6, 8, 9, 1, 8,
         0, 1, 0, 7, 0, 5, 5, 6, 1, 2, 0, 5, 0])
```

```
1 (index == y_test).sum()
```

```
tensor(319)
```

```
1 (index == y_test).sum() / len(y_test)
```

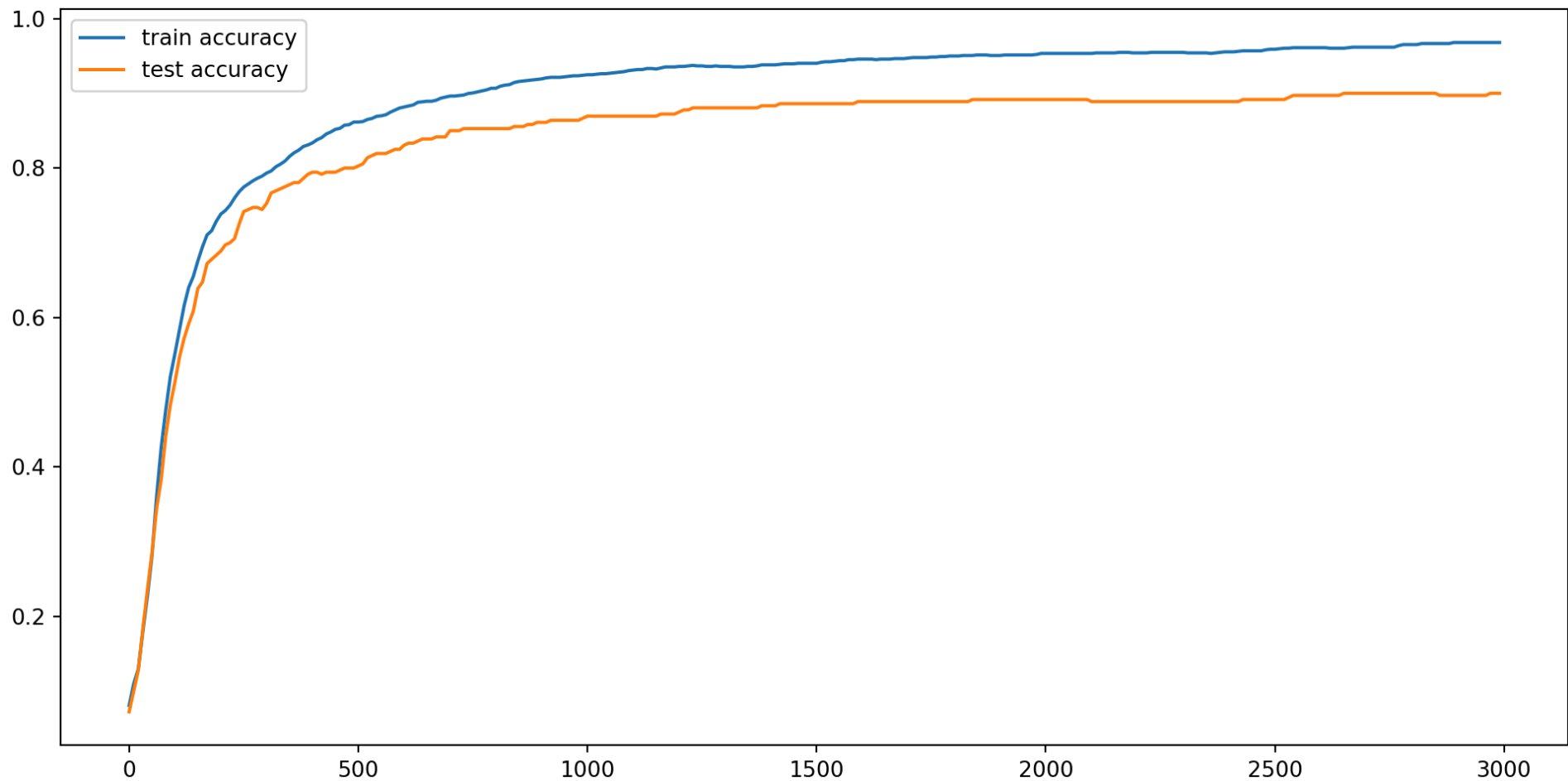
```
tensor(0.8861)
```

Calculating Accuracy

```
1 class mnist_model(torch.nn.Module):
2     def __init__(self, input_dim, output_dim):
3         super().__init__()
4         self.beta = torch.nn.Parameter(
5             torch.randn(input_dim, output_dim, requires_grad=True)
6         )
7         self.intercept = torch.nn.Parameter(
8             torch.randn(output_dim, requires_grad=True)
9         )
10
11     def forward(self, X):
12         return (X @ self.beta + self.intercept).squeeze()
13
14     def fit(self, X_train, y_train, X_test, y_test, lr=0.001, n=1000, acc_step=10):
15         opt = torch.optim.SGD(self.parameters(), lr=lr, momentum=0.9)
16         losses, train_acc, test_acc = [], [], []
17
18         for i in range(n):
19             opt.zero_grad()
20             loss = torch.nn.CrossEntropyLoss()(self(X_train), y_train)
21             loss.backward()
```

Performance

```
1 loss, train_acc, test_acc = mnist_model(  
2     64, 10  
3 ).fit(  
4     X_train, y_train, X_test, y_test, acc_step=10, n=3000  
5 )
```



NN Layers

```
1 class mnist_nn_model(torch.nn.Module):
2     def __init__(self, input_dim, output_dim):
3         super().__init__()
4         self.linear = torch.nn.Linear(input_dim, output_dim)
5
6     def forward(self, X):
7         return self.linear(X)
8
9     def fit(self, X_train, y_train, X_test, y_test, lr=0.001, n=1000, acc_step=10):
10         opt = torch.optim.SGD(self.parameters(), lr=lr, momentum=0.9)
11         losses, train_acc, test_acc = [], [], []
12
13         for i in range(n):
14             opt.zero_grad()
15             loss = torch.nn.CrossEntropyLoss()(self(X_train), y_train)
16             loss.backward()
17             opt.step()
18             losses.append(loss.item())
19
20             if (i+1) % acc_step == 0:
21                 val, train_pred = torch.max(self(X_train), dim=1)
```

NN linear layer

Applies a linear transform to the incoming data (X):

$$y = XA^T + b$$

```
1 X.shape
```

```
(1797, 64)
```

```
1 model = mnist_nn_model(64, 10)
2 model.parameters()
```

```
<generator object Module.parameters at 0x30ee31b60>
```

```
1 list(model.parameters())[0].shape # A – weights (betas)
```

```
torch.Size([10, 64])
```

```
1 list(model.parameters())[1].shape # b – bias
```

```
torch.Size([10])
```

Performance

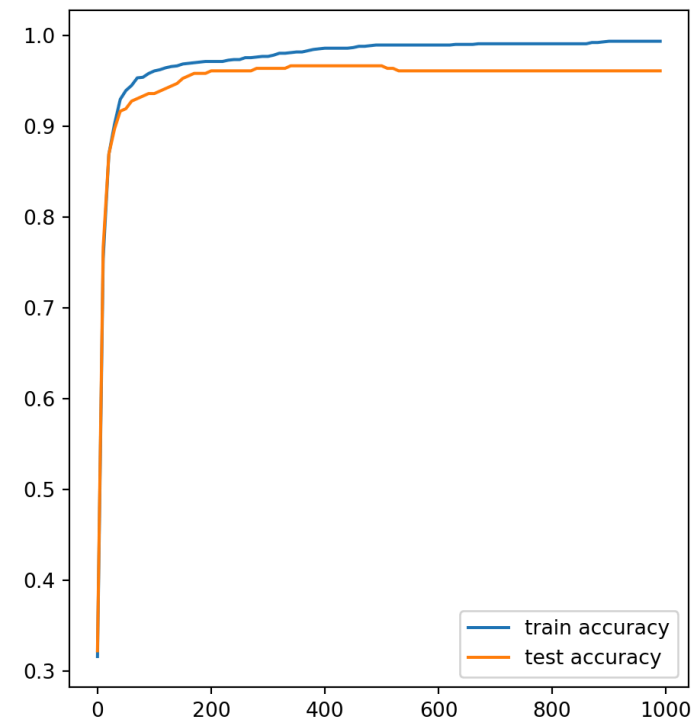
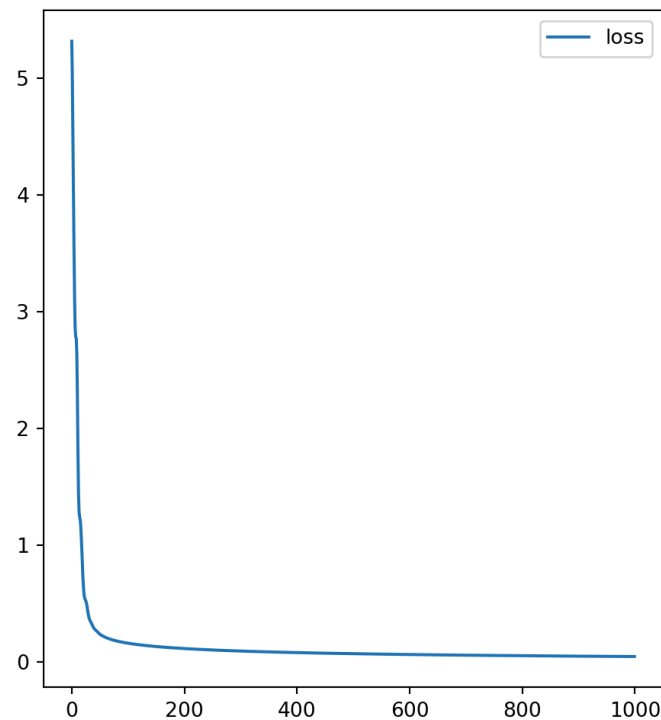
```
1 loss, train_acc, test_acc = model.fit(X_train, y_train, X_test, y_test, n=1000)
```

```
1 train_acc[-5:]
```

```
[tensor(0.9937), tensor(0.9937),  
tensor(0.9937), tensor(0.9937), tensor(0.9937)]
```

```
1 test_acc[-5:]
```

```
[tensor(0.9611), tensor(0.9611),  
tensor(0.9611), tensor(0.9611), tensor(0.9611)]
```



Feedforward Neural Network

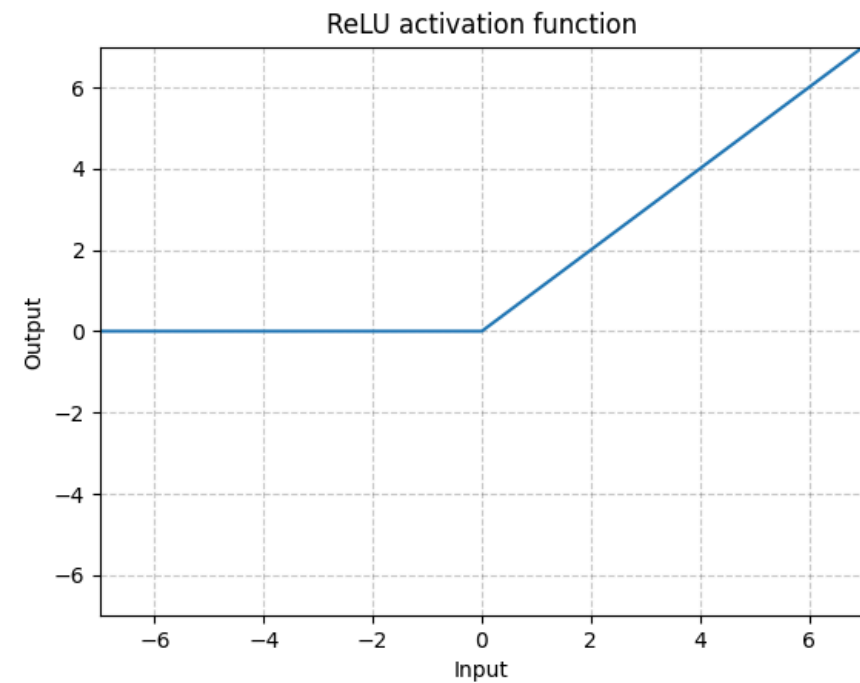
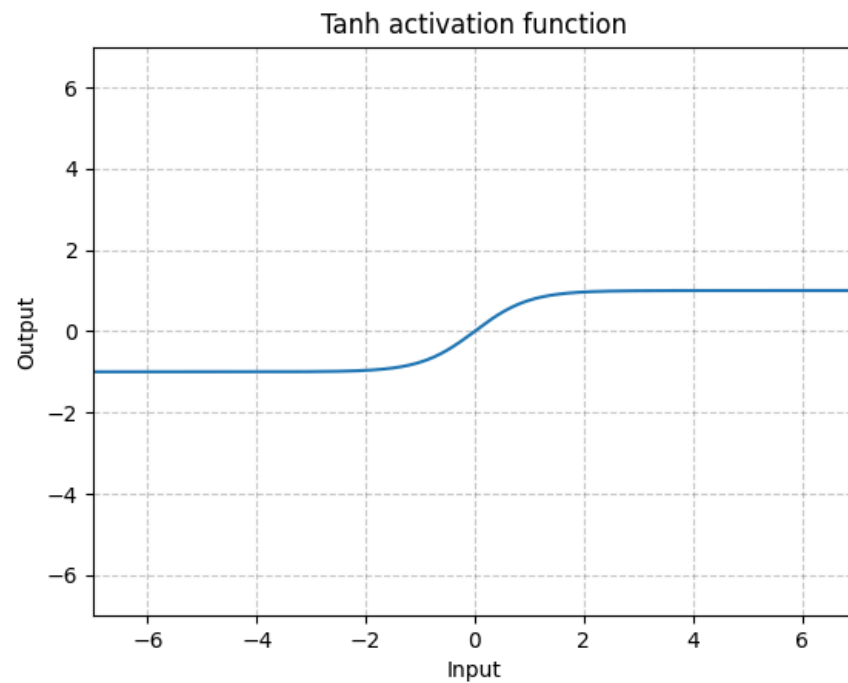
FNN Model

```
1 class mnist_fnn_model(torch.nn.Module):
2     def __init__(self, input_dim, hidden_dim, output_dim, nl_step = torch.nn.ReLU):
3         super().__init__()
4         self.l1 = torch.nn.Linear(input_dim, hidden_dim)
5         self.nl = nl_step
6         self.l2 = torch.nn.Linear(hidden_dim, output_dim)
7
8     def forward(self, X):
9         out = self.l1(X)
10        out = self.nl(out)
11        out = self.l2(out)
12        return out
13
14    def fit(self, X_train, y_train, X_test, y_test, lr=0.001, n=1000, acc_step=10):
15        opt = torch.optim.SGD(self.parameters(), lr=lr, momentum=0.9)
16        losses, train_acc, test_acc = [], [], []
17
18        for i in range(n):
19            opt.zero_grad()
20            loss = torch.nn.CrossEntropyLoss()(self(X_train), y_train)
21            loss.backward()
```

Non-linear activation functions

$$\text{Tanh}(x) = \frac{\exp(x) - \exp(-x)}{\exp(x) + \exp(-x)}$$

$$\text{ReLU}(x) = \max(0, x)$$



Model parameters

```
1 model = mnist_fnn_model(64, 64, 10)
2 len(list(model.parameters()))
```

4

```
1 for i, p in enumerate(model.parameters()):
2     print("Param", i, p.shape)
```

Param 0 torch.Size([64, 64])

Param 1 torch.Size([64])

Param 2 torch.Size([10, 64])

Param 3 torch.Size([10])

Performance - ReLU

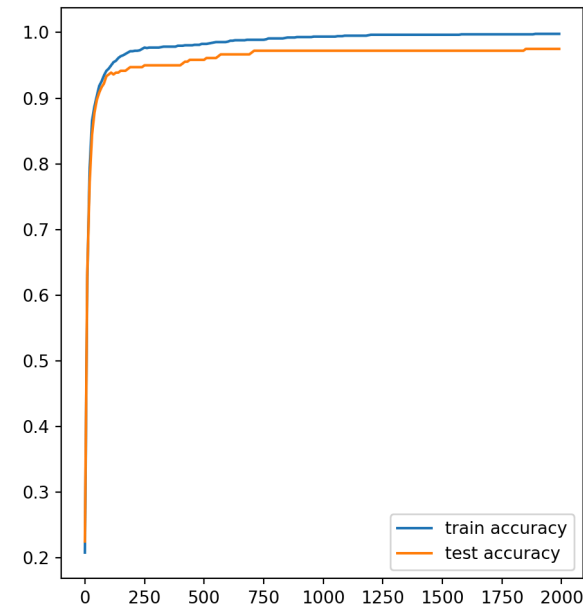
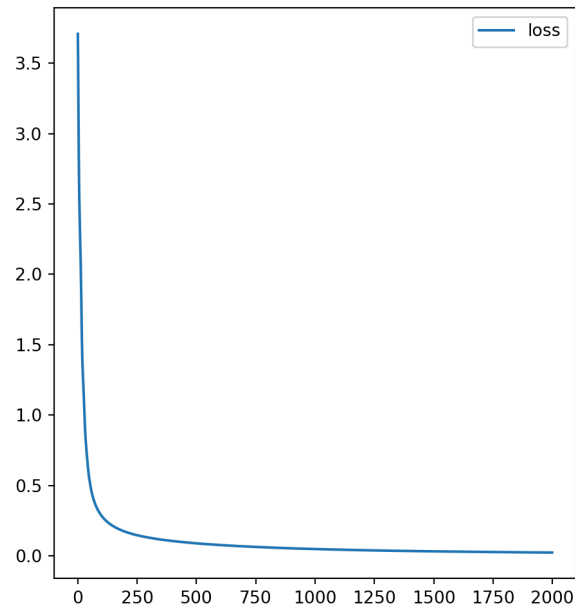
```
1 loss, train_acc, test_acc = mnist_fnn_model(64,64,10).fit(  
2   X_train, y_train, X_test, y_test, n=2000  
3   )
```

```
1 train_acc[-5:]
```

```
[0.9979123173277662, 0.9979123173277662,  
0.9979123173277662, 0.9979123173277662,  
0.9979123173277662]
```

```
1 test_acc[-5:]
```

```
[0.975, 0.975, 0.975, 0.975, 0.975]
```



Performance - tanh

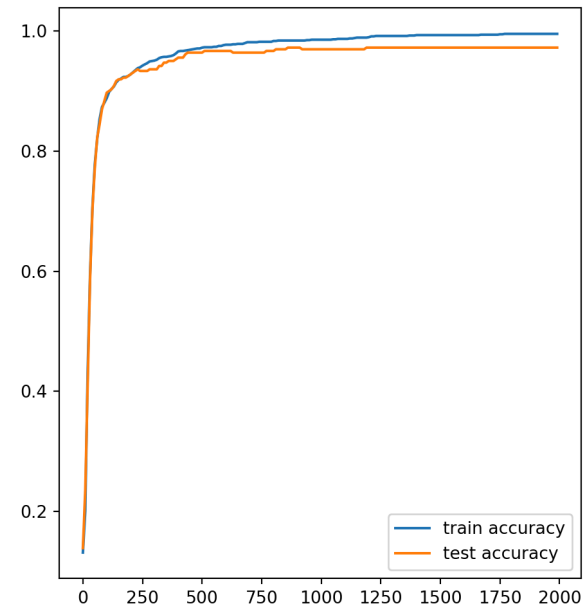
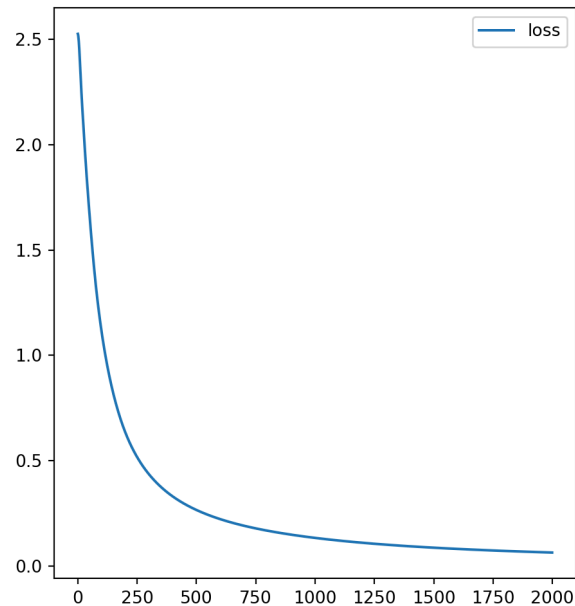
```
1 loss, train_acc, test_acc = mnist_fnn_model(64,64,10, nl_step=torch.nn.Tanh()).fit(  
2   X_train, y_train, X_test, y_test, n=2000  
3   )
```

```
1 train_acc[-5:]
```

```
[0.9951287404314544, 0.9951287404314544,  
0.9951287404314544, 0.9951287404314544,  
0.9951287404314544]
```

```
1 test_acc[-5:]
```

```
[0.9722222222222222, 0.9722222222222222,  
0.9722222222222222, 0.9722222222222222,  
0.9722222222222222]
```



Adding another layer

```
1 class mnist_fnn2_model(torch.nn.Module):
2     def __init__(self, input_dim, hidden_dim, output_dim, nl_step = torch.nn.ReLU(), seed=123):
3         super().__init__()
4         self.l1 = torch.nn.Linear(input_dim, hidden_dim)
5         self.nl = nl_step
6         self.l2 = torch.nn.Linear(hidden_dim, hidden_dim)
7         self.nl = nl_step
8         self.l3 = torch.nn.Linear(hidden_dim, output_dim)
9
10    def forward(self, X):
11        out = self.l1(X)
12        out = self.nl(out)
13        out = self.l2(out)
14        out = self.nl(out)
15        out = self.l3(out)
16        return out
17
18    def fit(self, X_train, y_train, X_test, y_test, lr=0.001, n=1000, acc_step=10):
19        loss_fn = torch.nn.CrossEntropyLoss()
20        opt = torch.optim.SGD(self.parameters(), lr=lr, momentum=0.9)
21        losses, train_acc, test_acc = [], [], []
22
23        for i in range(n):
24            opt.zero_grad()
```

Performance - relu

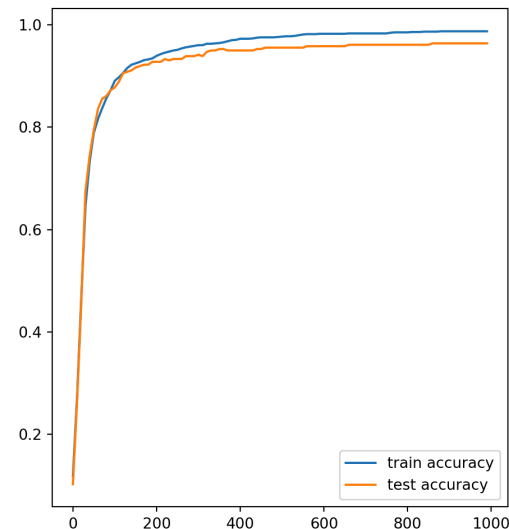
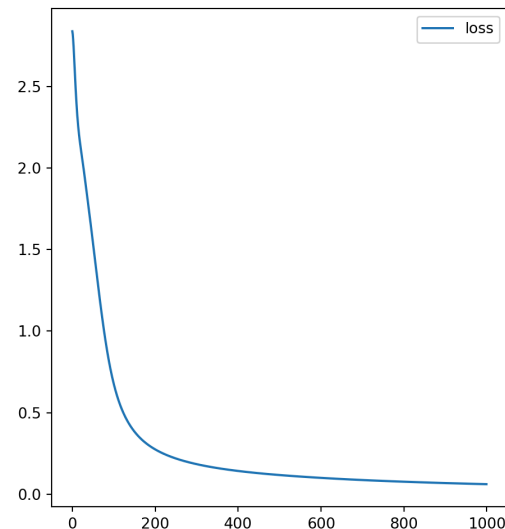
```
1 loss, train_acc, test_acc = mnist_fnn2_model(  
2     64,64,10, nl_step=torch.nn.ReLU()  
3 ).fit(  
4     X_train, y_train, X_test, y_test, n=1000  
5 )
```

```
1 train_acc[-5:]
```

```
[0.9874739039665971, 0.9874739039665971,  
0.9874739039665971, 0.9874739039665971,  
0.9874739039665971]
```

```
1 test_acc[-5:]
```

```
[0.9638888888888889, 0.9638888888888889,  
0.9638888888888889, 0.9638888888888889,  
0.9638888888888889]
```



Performance - tanh

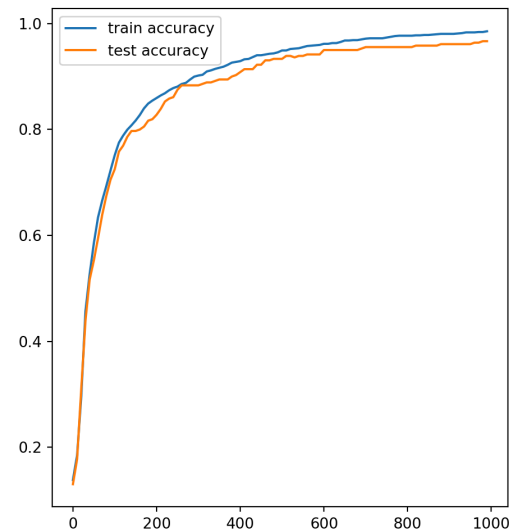
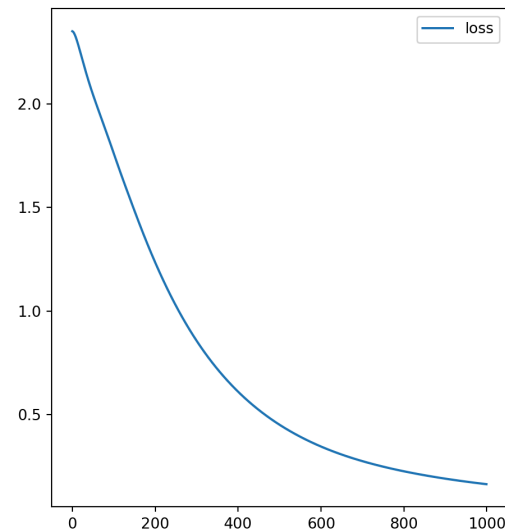
```
1 loss, train_acc, test_acc = mnist_fnn2_model(  
2     64,64,10, nl_step=torch.nn.Tanh()  
3 ).fit(  
4     X_train, y_train, X_test, y_test, n=1000  
5 )
```

```
1 train_acc[-5:]
```

```
[0.9832985386221295, 0.9832985386221295,  
0.9839944328462074, 0.9839944328462074,  
0.9853862212943633]
```

```
1 test_acc[-5:]
```

```
[0.9611111111111111, 0.9638888888888889,  
0.9638888888888889, 0.9666666666666667,  
0.9666666666666667]
```

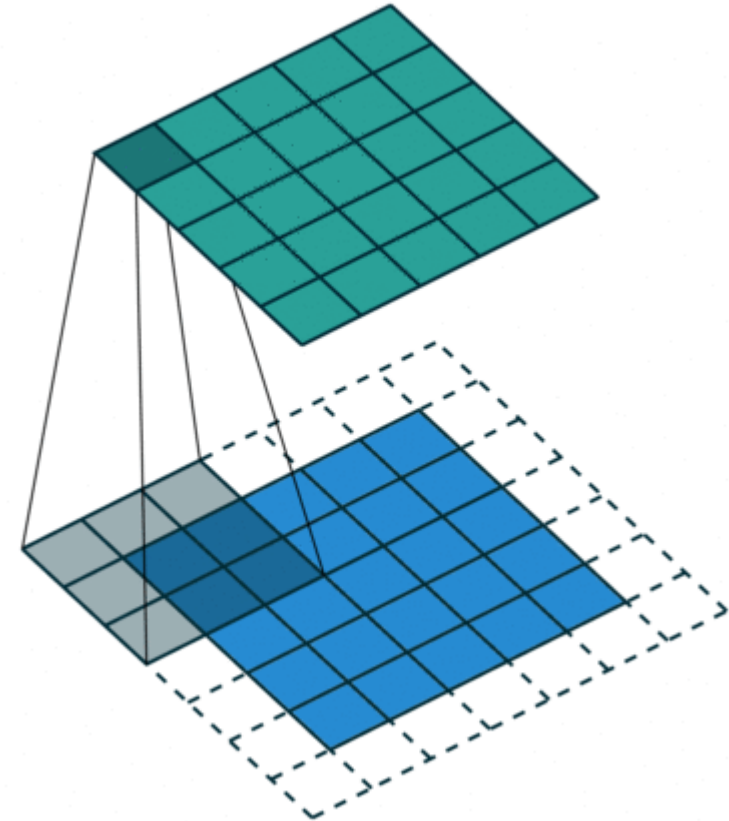


Convolutional NN

2d convolutions

3_0	3_1	2_2	1	0
0_2	0_2	1_0	3	1
3_0	1_1	2_2	2	3
2	0	0	2	2
2	0	0	0	1

12.0	12.0	17.0
10.0	17.0	19.0
9.0	6.0	14.0



nn.Conv2d()

```
1 cv = torch.nn.Conv2d(  
2     in_channels=1, out_channels=4,  
3     kernel_size=3,  
4     stride=1, padding=1  
5 )
```

```
1 list(cv.parameters())[0] # kernel weights
```

Parameter containing:

```
tensor([[[[ 1.8830e-03, -1.4839e-02,  
            8.2560e-03],  
          [ 3.2457e-01,  3.1790e-01,  
            -9.0510e-03],  
          [ 2.2771e-01,  2.3472e-01,  
            -2.7522e-01]]],  
        [[[ 1.8235e-01,  1.6332e-05,  
            3.3145e-01],  
          [ 2.4308e-01,  5.7604e-03,  
            -2.4588e-01],  
          [-2.6864e-01,  2.9254e-01,  
            2.4063e-01]]],  
        [[[-9.5288e-02, -2.5502e-01,  
            -2.0350e-01],  
          [ 7.5408e-02, -2.0131e-02,
```

```
1 list(cv.parameters())[1] # biases
```

Parameter containing:

```
tensor([-0.2965, -0.1130,  0.1528,  0.1705],  
        requires_grad=True)
```

Applying Conv2d ()


```
1 X_train[[0]]
```

```
tensor([[ 0.,  0.,  0., 10., 11.,  0.,  0.,
          0.,  0.,  0.,  9., 16.,  6.,  0.,
          0.,  0.,  0.,  0., 15., 13.,  0.,
          0.,  0.,  0.,  0.,  0., 14., 10.,
          0.,  0.,  0.,  0.,  0.,  1., 15.,
         12.,  8.,  2.,  0.,  0.,  0.,  0.,
         12., 16., 16., 16., 10.,  1.,  0.,
          0.,  7., 16., 12., 12., 16.,  4.,
          0.,  0.,  0.,  9., 15., 12.,  5.,
          0.]])
```

```
1 X_train[[0]].shape
```

```
torch.Size([1, 64])
```

```
1 cv(X_train[[0]])
```

RuntimeError: Expected 3D (unbatched) or 4D (batched) input to conv2d, but got input of size: [1, 64]

```
1 X_train[[0]].view(1,8,8)
```

```
tensor([[[ 0.,  0.,  0., 10., 11.,  0.,  0.,
           0.],
          [ 0.,  0.,  9., 16.,  6.,  0.,  0.,
           0.],
          [ 0.,  0., 15., 13.,  0.,  0.,  0.,
           0.],
          [ 0.,  0., 14., 10.,  0.,  0.,  0.,
           0.],
          [ 0.,  1., 15., 12.,  8.,  2.,  0.,
           0.],
          [ 0.,  0., 12., 16., 16., 16., 10.,
           1.],
          [ 0.,  0.,  7., 16., 12., 12., 16.,
           4.],
          [ 0.,  0.,  0.,  9., 15., 12.,  5.,
           0.]])])
```

```
1 cv(X_train[[0]].view(1,8,8))
```

```
tensor([[[ -0.2965, -2.7735, -2.6780,
            6.9365, 11.4978,  4.6400,
           -0.2965, -0.2965],
          [ -0.2965, -4.5062,  2.4452,
            14.0662,  9.6199,  1.6716,
           -0.2965, -0.2965],
          [ -0.2965, -4.2110,  4.8867,
            14.0689,  6.1411, -0.2852,
           -0.2965, -0.2965],
          [ -0.5717, -4.1929,  4.3942,
            11.2924,  7.0335,  1.9946,
            0.1589, -0.2965],
          [ -0.3056, -3.3014,  2.9759,
```

Pooling

```
1 x = torch.tensor(  
2     [[0,0,0,0],  
3      [0,1,2,0],  
4      [0,3,4,0],  
5      [0,0,0,0]]],  
6     dtype=torch.float  
7 )  
8 x.shape
```

torch.Size([1, 4, 4])

```
1 torch.nn.MaxPool2d(  
2     kernel_size=2, stride=1  
3 )(x)
```

tensor([[[1., 2., 2.],
 [3., 4., 4.],
 [3., 4., 4.]]])

```
1 torch.nn.MaxPool2d(  
2     kernel_size=3, stride=1, padding=1  
3 )(x)
```

tensor([[[1., 2., 2., 2.],
 [3., 4., 4., 4.],
 [3., 4., 4., 4.],
 [3., 4., 4., 4.]]])

```
1 torch.nn.AvgPool2d(  
2     kernel_size=2  
3 )(x)
```

tensor([[[0.2500, 0.5000],
 [0.7500, 1.0000]])])

```
1 torch.nn.AvgPool2d(  
2     kernel_size=2, padding=1  
3 )(x)
```

tensor([[[0.0000, 0.0000, 0.0000],
 [0.0000, 2.5000, 0.0000],
 [0.0000, 0.0000, 0.0000]])])

Convolutional model

```
1 class mnist_conv_model(torch.nn.Module):
2     def __init__(self):
3         super().__init__()
4         self.cnn = torch.nn.Conv2d(
5             in_channels=1, out_channels=8,
6             kernel_size=3, stride=1, padding=1
7         )
8         self.relu = torch.nn.ReLU()
9         self.pool = torch.nn.MaxPool2d(kernel_size=2)
10        self.lin = torch.nn.Linear(8 * 4 * 4, 10)
11
12    def forward(self, X):
13        out = self.cnn(X.view(-1, 1, 8, 8))
14        out = self.relu(out)
15        out = self.pool(out)
16        out = self.lin(out.view(-1, 8 * 4 * 4))
17        return out
18
19    def fit(self, X_train, y_train, X_test, y_test, lr=0.001, n=1000, acc_step=10)
20        loss_fn = torch.nn.CrossEntropyLoss()
21        opt = torch.optim.SGD(self.parameters(), lr=lr, momentum=0.9)
```

Performance

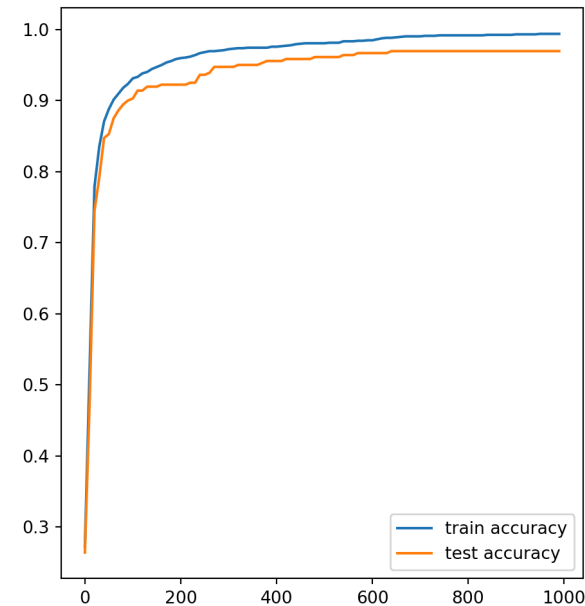
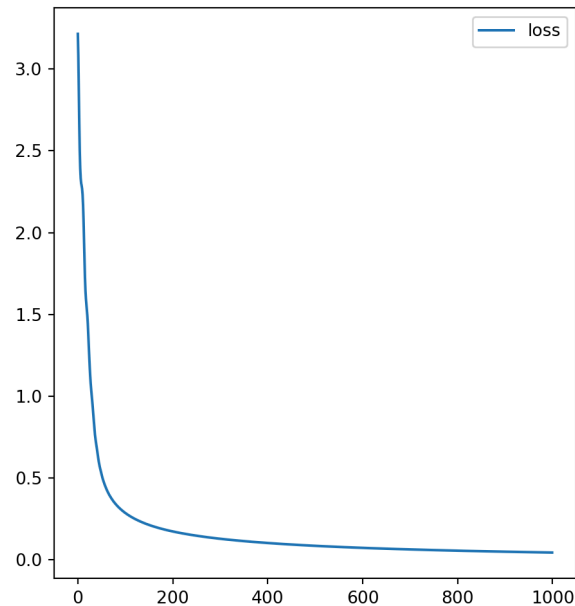
```
1 loss, train_acc, test_acc = mnist_conv_model().fit(  
2   X_train, y_train, X_test, y_test, n=1000  
3 )
```

```
1 train_acc[-5:]
```

```
[0.9937369519832986, 0.9937369519832986,  
0.9937369519832986, 0.9937369519832986,  
0.9937369519832986]
```

```
1 test_acc[-5:]
```

```
[0.9694444444444444, 0.9694444444444444,  
0.9694444444444444, 0.9694444444444444,  
0.9694444444444444]
```



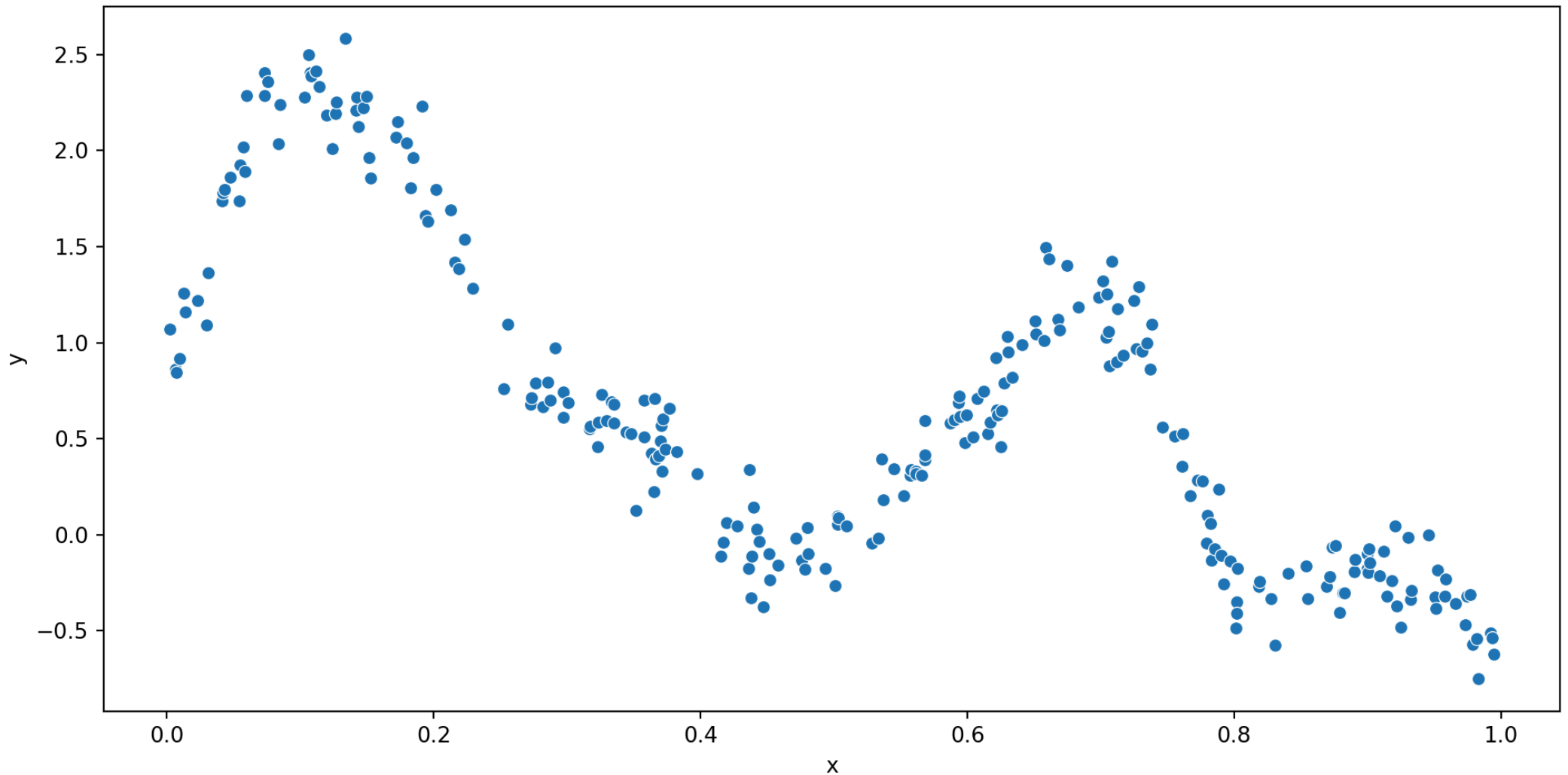
Organizing models

```
1 class mnist_conv_model2(torch.nn.Module):
2     def __init__(self):
3         super().__init__()
4         self.model = torch.nn.Sequential(
5             torch.nn.Unflatten(1, (1,8,8)),
6             torch.nn.Conv2d(
7                 in_channels=1, out_channels=8,
8                 kernel_size=3, stride=1, padding=1
9             ),
10            torch.nn.ReLU(),
11            torch.nn.MaxPool2d(kernel_size=2),
12            torch.nn.Flatten(),
13            torch.nn.Linear(8 * 4 * 4, 10)
14        )
15
16    def forward(self, X):
17        return self.model(X)
18
19    def fit(self, X_train, y_train, X_test, y_test, lr=0.001, n=1000, acc_step=10):
20        opt = torch.optim.SGD(self.parameters(), lr=lr, momentum=0.9)
21        losses, train_acc, test_acc = [], [], []
22
23        for i in range(n):
24            opt.zero_grad()
```

A bit more on non-linear activation layers

Non-linear functions

```
1 df = pd.read_csv("data/gp.csv")
2 X = torch.tensor(df["x"], dtype=torch.float32).reshape(-1,1)
3 y = torch.tensor(df["y"], dtype=torch.float32)
```



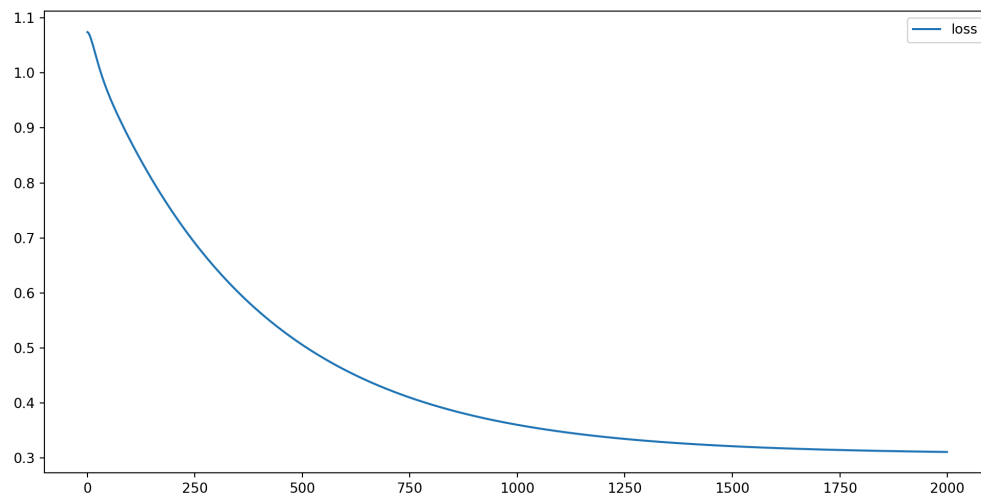
Linear regression

```
1 class lin_reg(torch.nn.Module):
2     def __init__(self, X):
3         super().__init__()
4         self.n = X.shape[0]
5         self.p = X.shape[1]
6         self.model = torch.nn.Sequential(
7             torch.nn.Linear(self.p, self.p)
8         )
9
10    def forward(self, X):
11        return self.model(X)
12
13    def fit(self, X, y, n=1000):
14        losses = []
15        opt = torch.optim.SGD(self.parameters(), lr=0.001, momentum=0.9)
16        for i in range(n):
17            loss = torch.nn.MSELoss()(self(X).squeeze(), y)
18            loss.backward()
19            opt.step()
20            opt.zero_grad()
21            losses.append(loss.item())
22
23        return losses
```

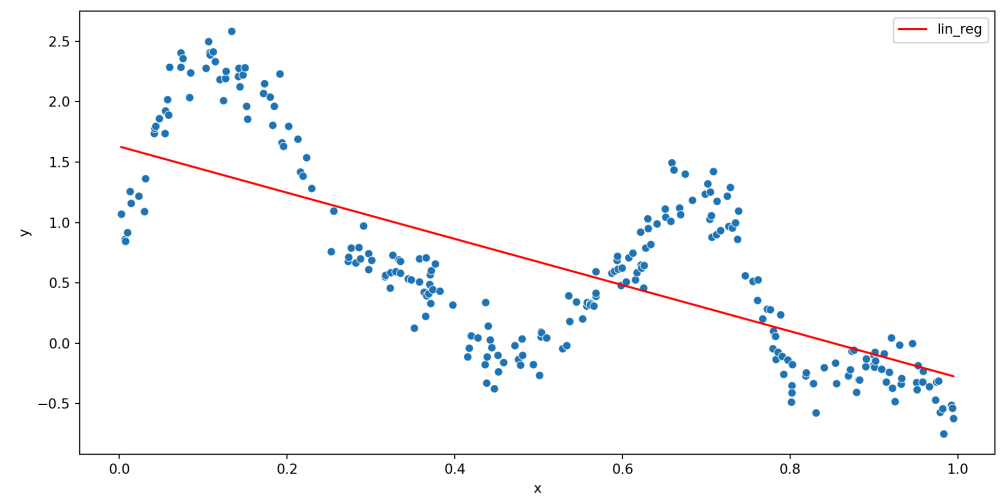

Model results

```
1 m1 = lin_reg(X)
2 loss = m1.fit(X,y, n=2000)
```

Training loss:



Predictions



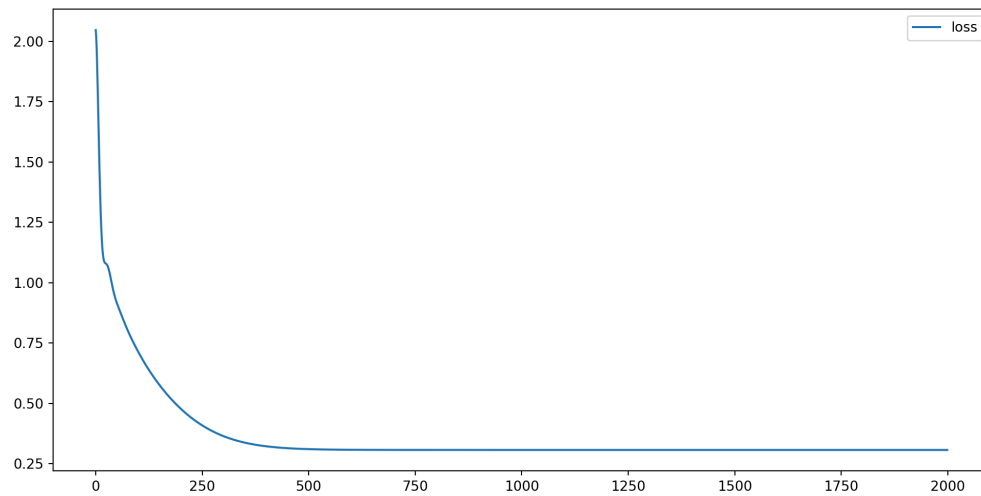
Double linear regression

```
1 class dbl_lin_reg(torch.nn.Module):
2     def __init__(self, X, hidden_dim=10):
3         super().__init__()
4         self.n = X.shape[0]
5         self.p = X.shape[1]
6         self.model = torch.nn.Sequential(
7             torch.nn.Linear(self.p, hidden_dim),
8             torch.nn.Linear(hidden_dim, 1)
9         )
10
11     def forward(self, X):
12         return self.model(X)
13
14     def fit(self, X, y, n=1000):
15         losses = []
16         opt = torch.optim.SGD(self.parameters(), lr=0.001, momentum=0.9)
17         for i in range(n):
18             loss = torch.nn.MSELoss()(self(X).squeeze(), y)
19             loss.backward()
20             opt.step()
21             opt.zero_grad()
22             losses.append(loss.item())
23
24         return losses
```

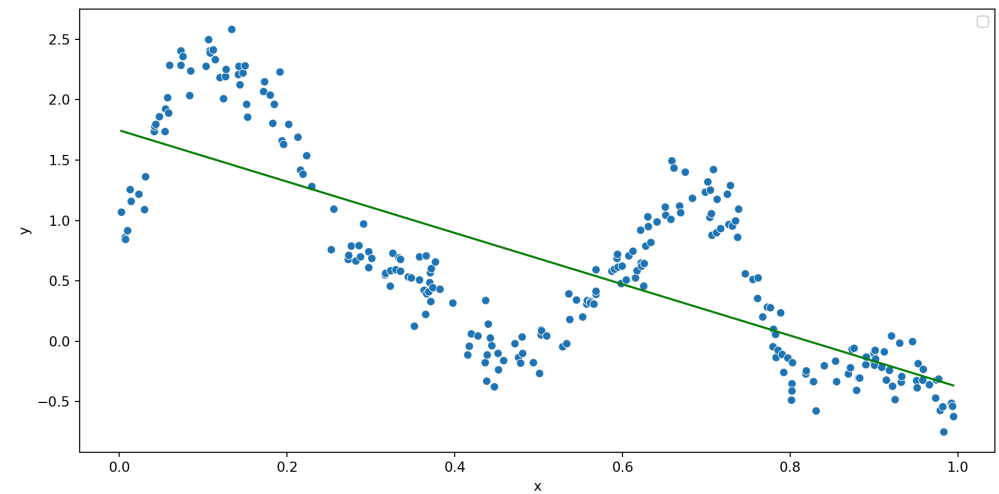
Model results

```
1 m2 = dbl_lin_reg(X, hidden_dim=10)
2 loss = m2.fit(X,y, n=2000)
```

Training loss:



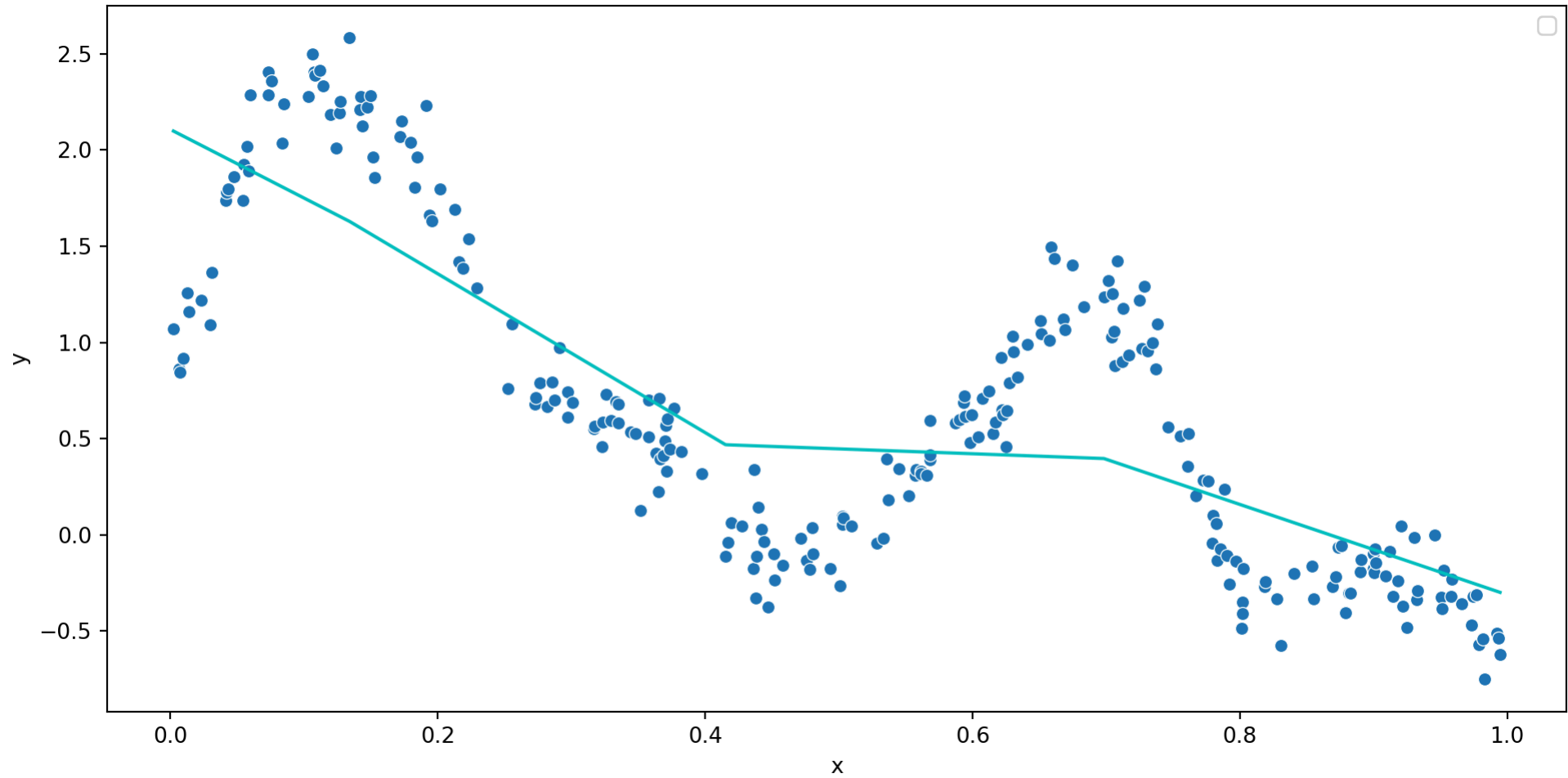
Predictions



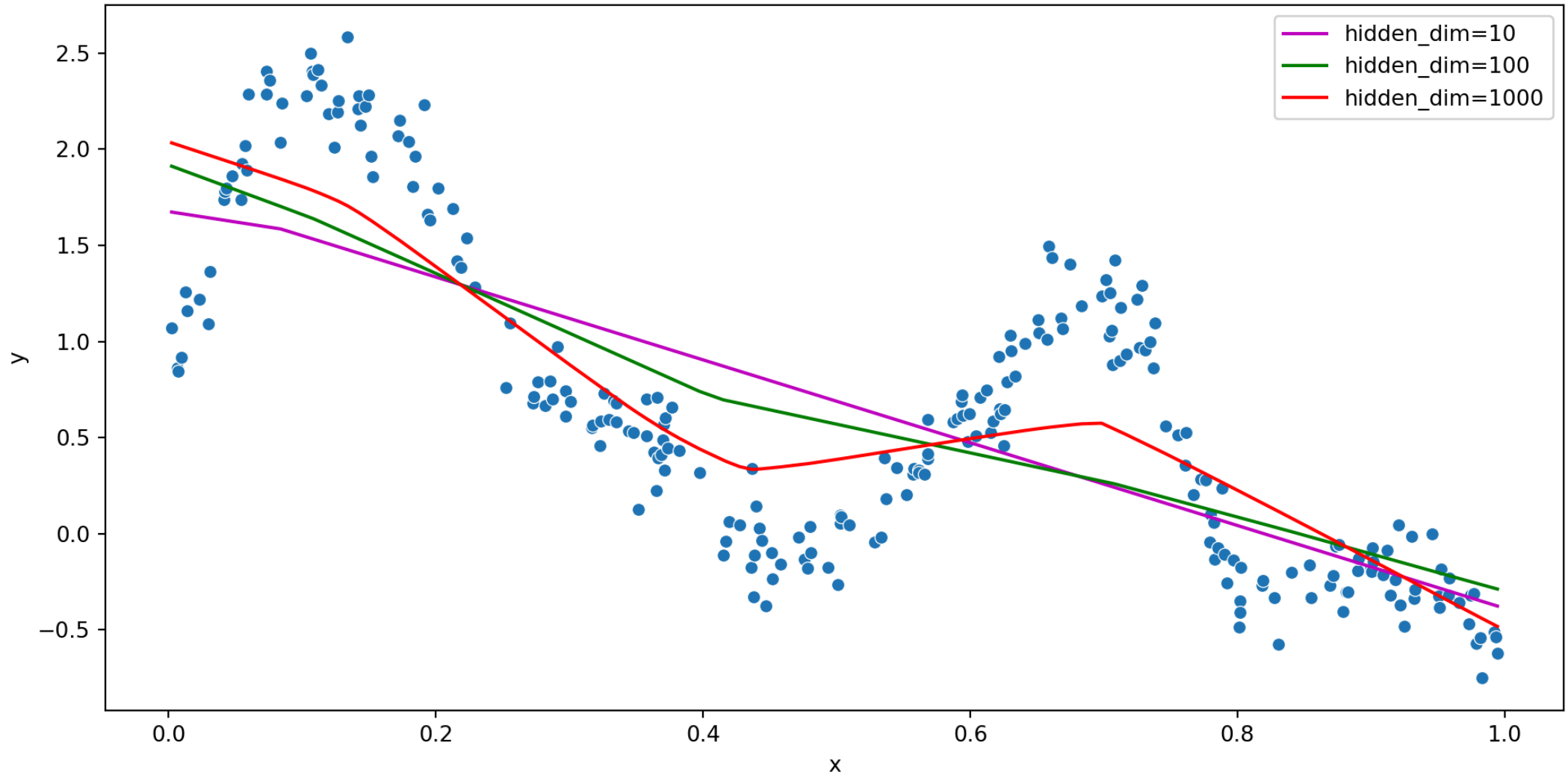
Non-linear regression w/ ReLU

```
1 class lin_reg_relu(torch.nn.Module):
2     def __init__(self, X, hidden_dim=100):
3         super().__init__()
4         self.n = X.shape[0]
5         self.p = X.shape[1]
6         self.model = torch.nn.Sequential(
7             torch.nn.Linear(self.p, hidden_dim),
8             torch.nn.ReLU(),
9             torch.nn.Linear(hidden_dim, 1)
10        )
11
12    def forward(self, X):
13        return self.model(X)
14
15    def fit(self, X, y, n=1000):
16        losses = []
17        opt = torch.optim.SGD(self.parameters(), lr=0.001, momentum=0.9)
18        for i in range(n):
19            loss = torch.nn.MSELoss()(self(X).squeeze(), y)
20            loss.backward()
21            opt.step()
22            opt.zero_grad()
23            losses.append(loss.item())
24
```

Model results



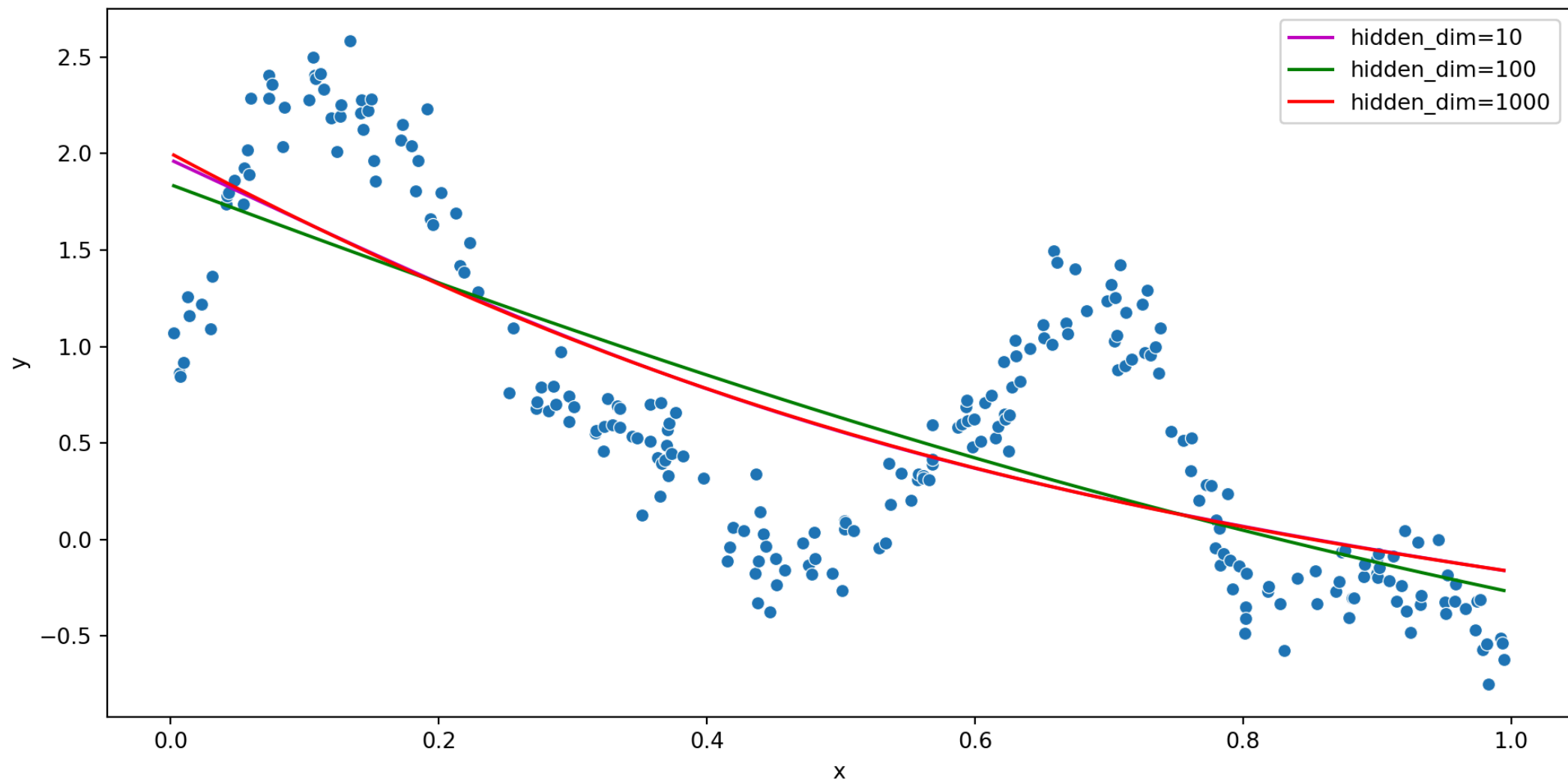
Hidden dimensions



Non-linear regression w/ Tanh

```
1 class lin_reg_tanh(torch.nn.Module):
2     def __init__(self, X, hidden_dim=10):
3         super().__init__()
4         self.n = X.shape[0]
5         self.p = X.shape[1]
6         self.model = torch.nn.Sequential(
7             torch.nn.Linear(self.p, hidden_dim),
8             torch.nn.Tanh(),
9             torch.nn.Linear(hidden_dim, 1)
10        )
11
12    def forward(self, X):
13        return self.model(X)
14
15    def fit(self, X, y, n=1000):
16        losses = []
17        opt = torch.optim.SGD(self.parameters(), lr=0.001, momentum=0.9)
18        for i in range(n):
19            loss = torch.nn.MSELoss()(self(X).squeeze(), y)
20            loss.backward()
21            opt.step()
22            opt.zero_grad()
23            losses.append(loss.item())
24
```

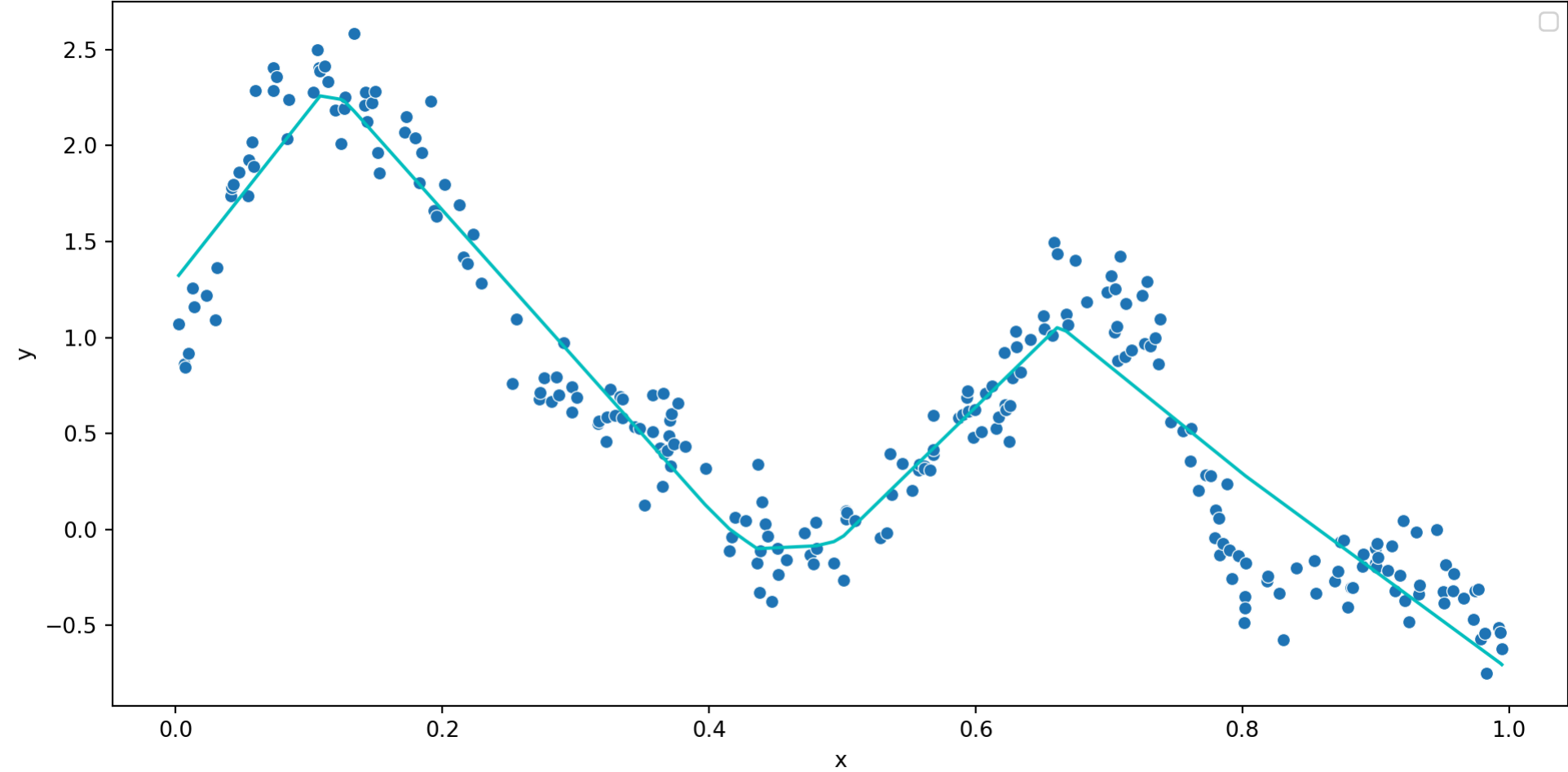
Tanh & hidden dimension



Three layers

```
1 class three_layers(torch.nn.Module):
2     def __init__(self, X, hidden_dim=100):
3         super().__init__()
4         self.n = X.shape[0]
5         self.p = X.shape[1]
6         self.model = torch.nn.Sequential(
7             torch.nn.Linear(self.p, hidden_dim),
8             torch.nn.ReLU(),
9             torch.nn.Linear(hidden_dim, hidden_dim),
10            torch.nn.ReLU(),
11            torch.nn.Linear(hidden_dim, 1)
12        )
13
14    def forward(self, X):
15        return self.model(X)
16
17    def fit(self, X, y, n=1000):
18        losses = []
19        opt = torch.optim.SGD(self.parameters(), lr=0.001, momentum=0.9)
20        for i in range(n):
21            loss = torch.nn.MSELoss()(self(X).squeeze(), y)
22            loss.backward()
23            opt.step()
24            opt.zero_grad()
```

Model results



Five layers

```
1 class five_layers(torch.nn.Module):
2     def __init__(self, X, hidden_dim=100):
3         super().__init__()
4         self.n = X.shape[0]
5         self.p = X.shape[1]
6         self.model = torch.nn.Sequential(
7             torch.nn.Linear(self.p, hidden_dim),
8             torch.nn.ReLU(),
9             torch.nn.Linear(hidden_dim, hidden_dim),
10            torch.nn.ReLU(),
11            torch.nn.Linear(hidden_dim, hidden_dim),
12            torch.nn.ReLU(),
13            torch.nn.Linear(hidden_dim, hidden_dim),
14            torch.nn.ReLU(),
15            torch.nn.Linear(hidden_dim, 1)
16        )
17
18    def forward(self, X):
19        return self.model(X)
20
21    def fit(self, X, y, n=1000):
22        losses = []
23        opt = torch.optim.SGD(self.parameters(), lr=0.001, momentum=0.9)
24        for i in range(n):
```

Model results

