pytorch - optim & nn

Lecture 23

Dr. Colin Rundel

Odds & Ends

Torch models

Implementation details:

- Models are implemented as a class - inherits 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 -> predictions

Fitting proceedure:

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

From last time

```
class Model(torch.nn.Module):
    def __init__(self, X, y, beta=None):
        super().__init__()
        self.X = X
        self.y = y
        if beta is None:
          beta = torch.zeros(X.shape[1])
        beta.requires_grad = True
        self.beta = torch.nn.Parameter(beta)

    def forward(self, X):
        return X @ self.beta

    def fit(self, opt, n=1000, loss_fn = torch.nn.MSELoss()):
      losses = []
      for i in range(n):
          loss = loss_fn(
            self(self.X).squeeze(),
            self.y.squeeze()
          )
          loss.backward()
          opt.step()
          opt.zero_grad()
```

What is `self(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 special `__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'
```


Using a class like a function is often refered to as a functor in other languages

MNIST & Logistic models

MNIST handwritten digits - simplified

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

```
X = digits.data
X.shape
```

```
(1797, 64)
```

```
y = digits.target
y.shape
```

```
(1797,)
```

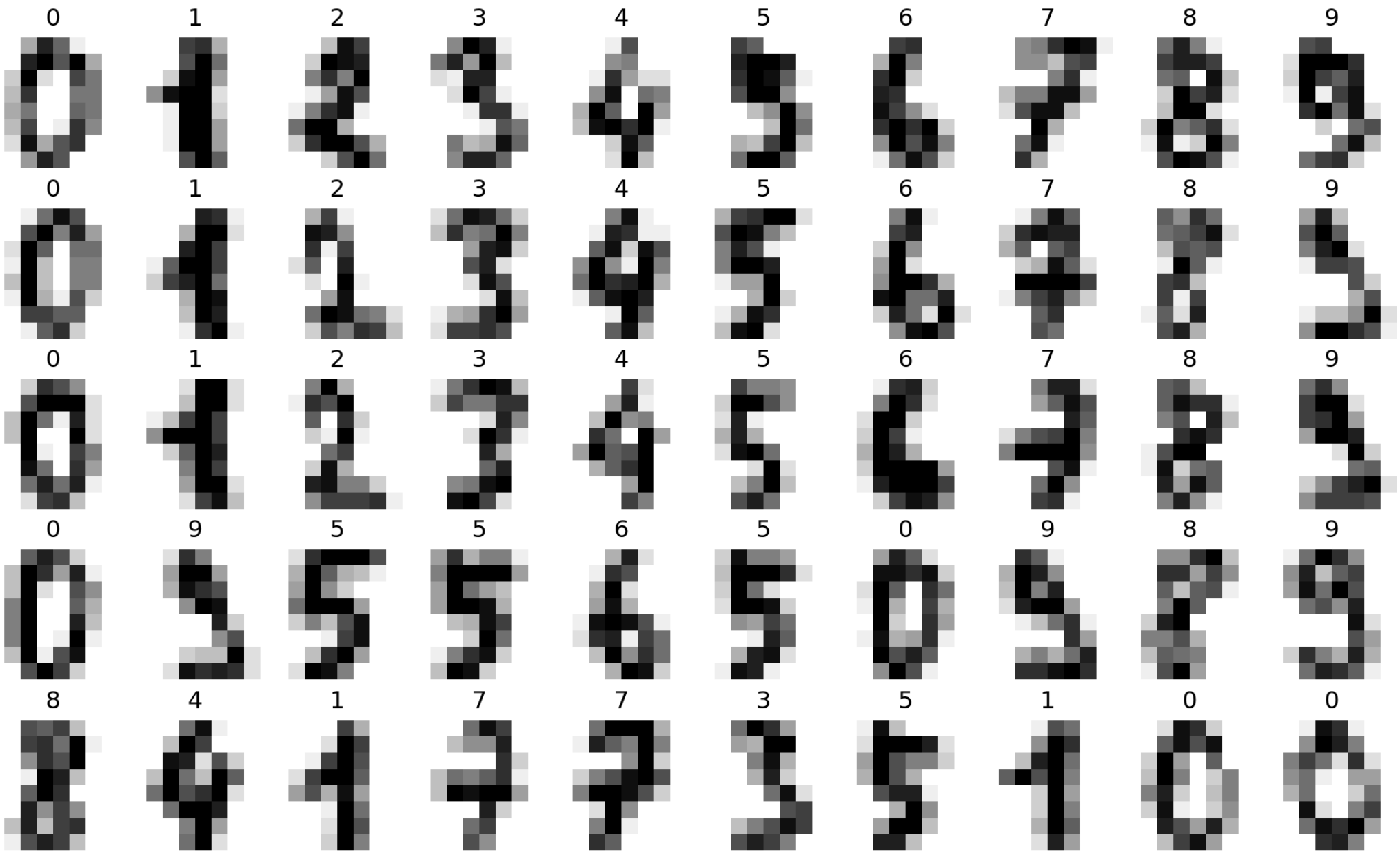
```
X[0:2]
```

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

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

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

Example digits



Test train split

```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.20, shuffle=True, random_state=1234
)
```

```
X_train.shape
```

```
(1437, 64)
```

```
y_train.shape
```

```
(1437,)
```

```
X_test.shape
```

```
(360, 64)
```

```
y_test.shape
```

```
(360,)
```

```
from sklearn.linear_model import LogisticRegress
from sklearn.metrics import accuracy_score
```

```
lr = LogisticRegression(
  penalty=None
).fit(
  X_train, y_train
)
```

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

```
1.0
```

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

```
0.9583333333333334
```

As Tensors

```
X_train = torch.from_numpy(X_train).float()
y_train = torch.from_numpy(y_train)
X_test = torch.from_numpy(X_test).float()
y_test = torch.from_numpy(y_test)
```

```
X_train.shape
```

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

```
y_train.shape
```

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

```
X_test.shape
```

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

```
y_test.shape
```

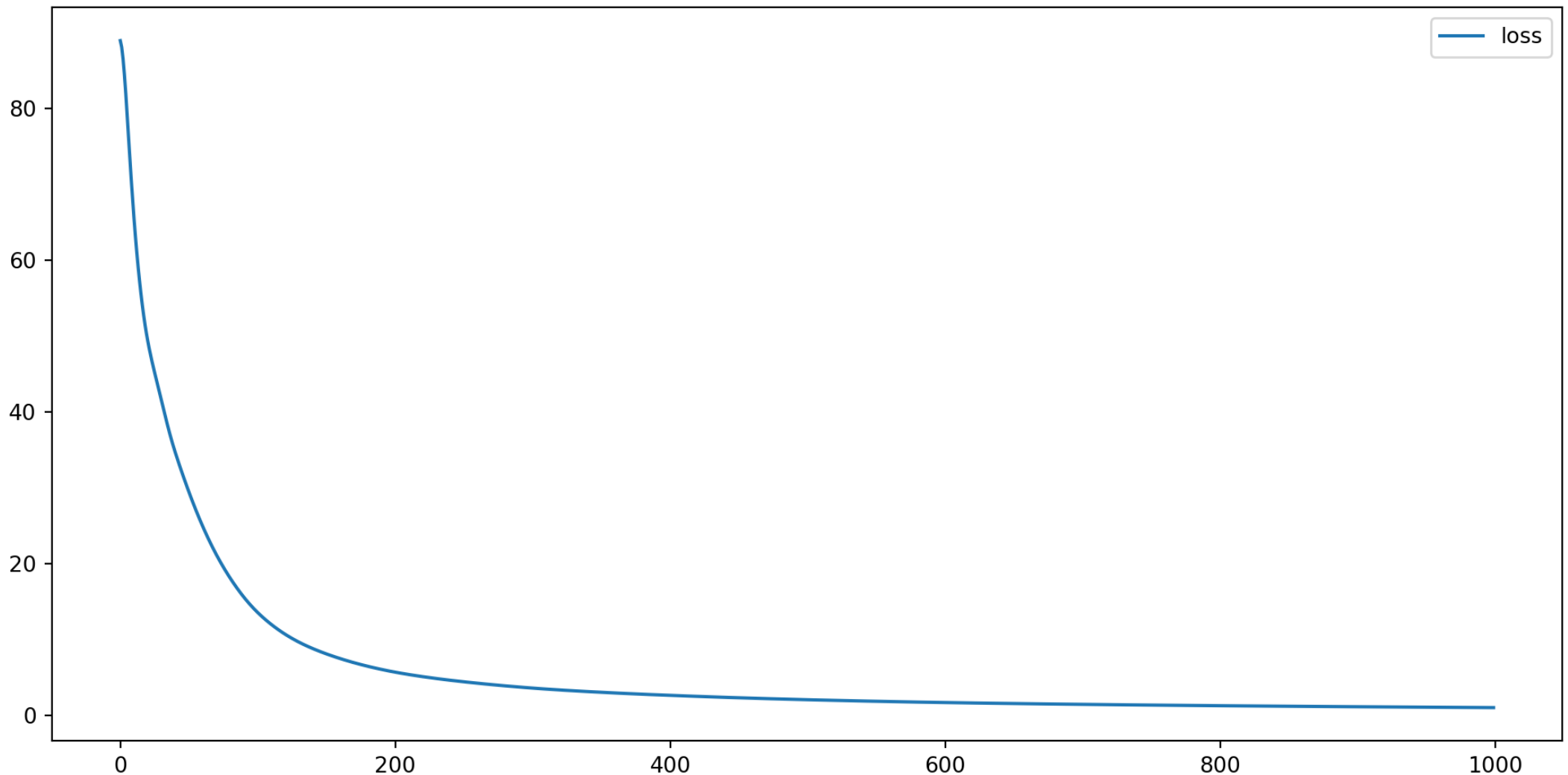
```
torch.Size([360])
```

PyTorch Model

```
class mnist_model(torch.nn.Module):
    def __init__(self, input_dim, output_dim):
        super().__init__()
        self.beta = torch.nn.Parameter(
          torch.randn(input_dim, output_dim, requires_grad=True)
        )
        self.intercept = torch.nn.Parameter(
          torch.randn(output_dim, requires_grad=True)
        )

    def forward(self, X):
        return (X @ self.beta + self.intercept).squeeze()

    def fit(self, X_train, y_train, X_test, y_test, lr=0.001, n=1000):
      opt = torch.optim.SGD(self.parameters(), lr=lr, momentum=0.9)
      losses = []

      for i in range(n):
          opt.zero_grad()
          loss = torch.nn.CrossEntropyLoss()(self(X_train), y_train)
          loss.backward()
          opt.step()

```

Cross entropy loss

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



Out-of-sample accuracy

```
model(X_test)
```

```
tensor([[-12.0853, -12.4920, -16.3669,  ...,  96.228
        [-41.2113,  26.0970,  -6.4925,  ..., -12.430
        [-32.5302,  11.3226,  13.1565,  ..., 112.713
        ...,
        [ 19.0148,  15.7732, -21.7103,  ...,  -3.028
        [-37.8395,  28.8395, -19.3067,  ...,  53.392
        [-34.6293,  -8.0961,  19.3990,  ...,  -7.284
       grad_fn=<SqueezeBackward0>)
```

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

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

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

```
tensor(313)
```

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

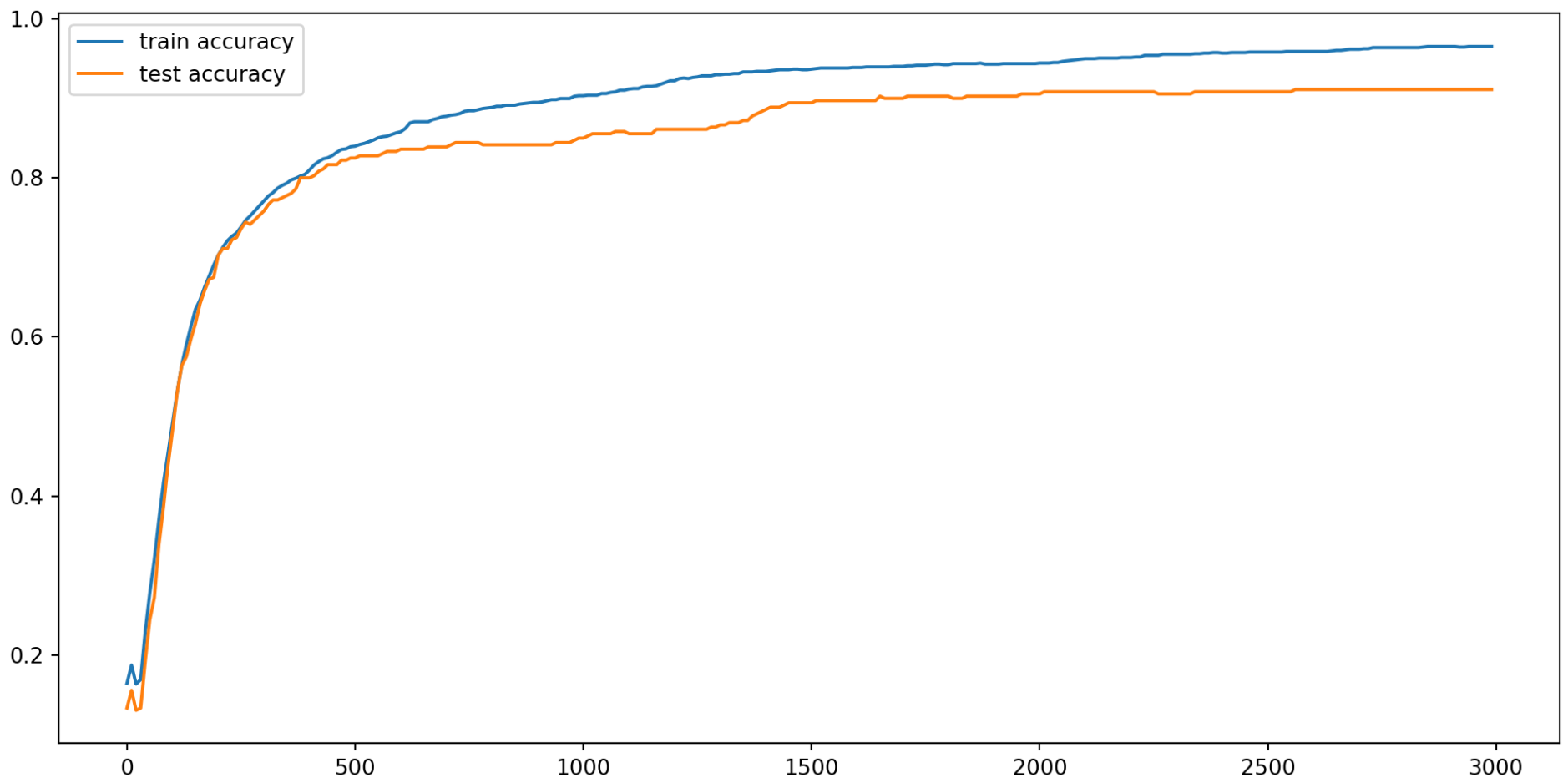
```
tensor(0.8694)
```

Calculating Accuracy

```
class mnist_model(torch.nn.Module):
    def __init__(self, input_dim, output_dim):
        super().__init__()
        self.beta = torch.nn.Parameter(
          torch.randn(input_dim, output_dim, requires_grad=True)
        )
        self.intercept = torch.nn.Parameter(
          torch.randn(output_dim, requires_grad=True)
        )

    def forward(self, X):
        return (X @ self.beta + self.intercept).squeeze()

    def fit(self, X_train, y_train, X_test, y_test, lr=0.001, n=1000, acc_step=10):
      opt = torch.optim.SGD(self.parameters(), lr=lr, momentum=0.9)
      losses, train_acc, test_acc = [], [], []

      for i in range(n):
          opt.zero_grad()
          loss = torch.nn.CrossEntropyLoss()(self(X_train), y_train)
          loss.backward()
          opt.step()
          losses.append(loss.item())
```

Performance

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



NN Layers

```
class mnist_nn_model(torch.nn.Module):
    def __init__(self, input_dim, output_dim):
        super().__init__()
        self.linear = torch.nn.Linear(input_dim, output_dim)

    def forward(self, X):
        return self.linear(X)

    def fit(self, X_train, y_train, X_test, y_test, lr=0.001, n=1000, acc_step=10):
      opt = torch.optim.SGD(self.parameters(), lr=lr, momentum=0.9)
      losses, train_acc, test_acc = [], [], []

      for i in range(n):
          opt.zero_grad()
          loss = torch.nn.CrossEntropyLoss()(self(X_train), y_train)
          loss.backward()
          opt.step()
          losses.append(loss.item())

          if (i+1) % acc_step == 0:
            val, train_pred = torch.max(self(X_train), dim=1)
            val, test_pred  = torch.max(self(X_test), 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 0x2c19d9700>
```

```
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

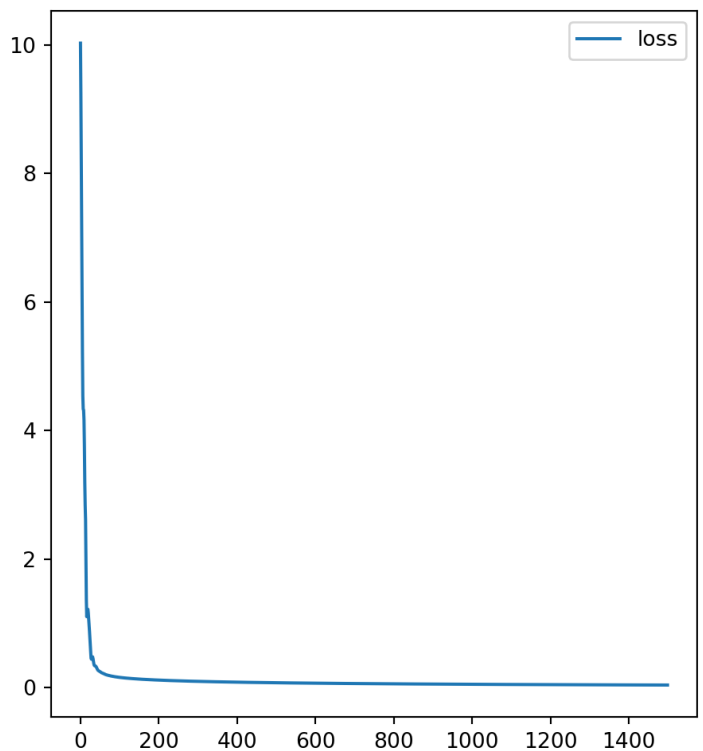
```
1 loss, train_acc, test_acc = model.fit(X_train, y_train, X_test, y_test, n=1500)
```

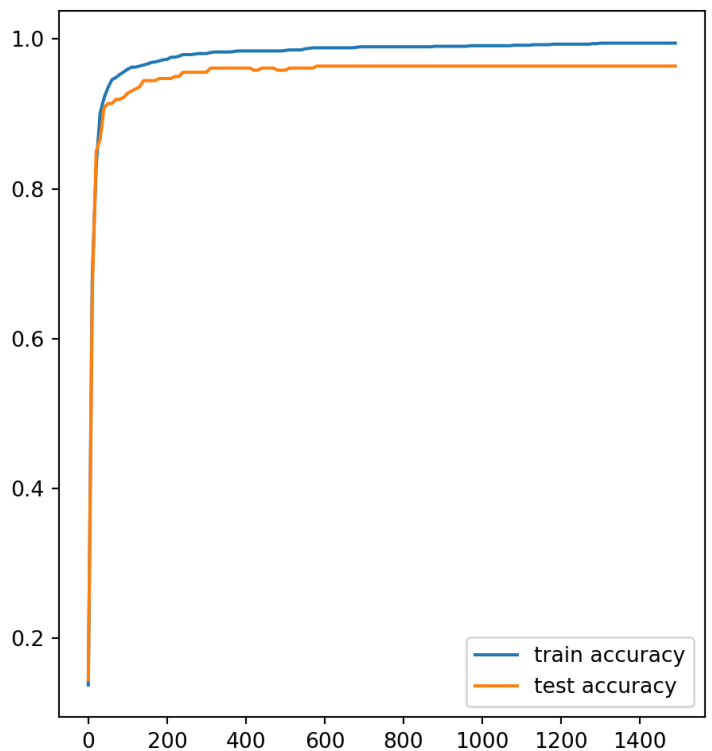
```
1 train_acc[-5:]
```

```
[tensor(0.9944), tensor(0.9944), tensor(0.9944), tens
```

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

```
[tensor(0.9639), tensor(0.9639), tensor(0.9639), tens
```





Feedforward Neural Network

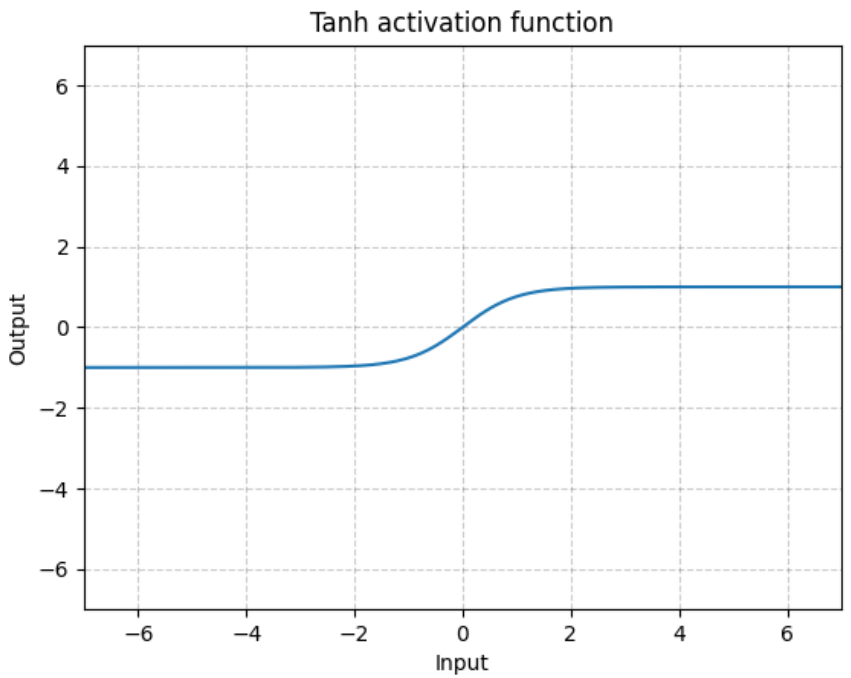
FNN Model

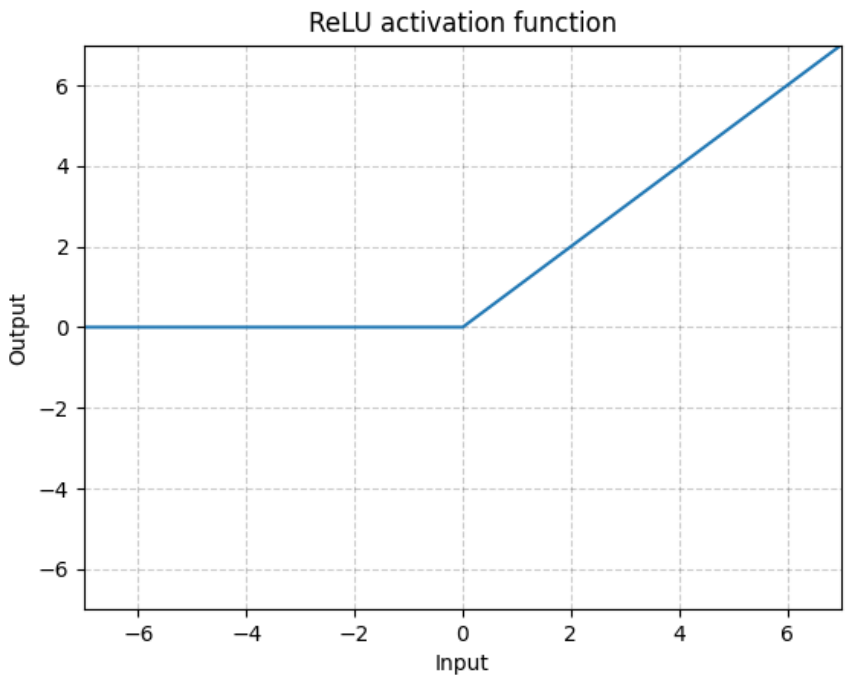
```
class mnist_fnn_model(torch.nn.Module):
    def __init__(self, input_dim, hidden_dim, output_dim, nl_step = torch.nn.ReLU(), seed=1234):
        super().__init__()
        self.l1 = torch.nn.Linear(input_dim, hidden_dim)
        self.nl = nl_step
        self.l2 = torch.nn.Linear(hidden_dim, output_dim)

    def forward(self, X):
        out = self.l1(X)
        out = self.nl(out)
        out = self.l2(out)
        return out

    def fit(self, X_train, y_train, X_test, y_test, lr=0.001, n=1000, acc_step=10):
      opt = torch.optim.SGD(self.parameters(), lr=lr, momentum=0.9)
      losses, train_acc, test_acc = [], [], []

      for i in range(n):
          opt.zero_grad()
          loss = torch.nn.CrossEntropyLoss()(self(X_train), y_train)
          loss.backward()
          opt.step()

```

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])
```

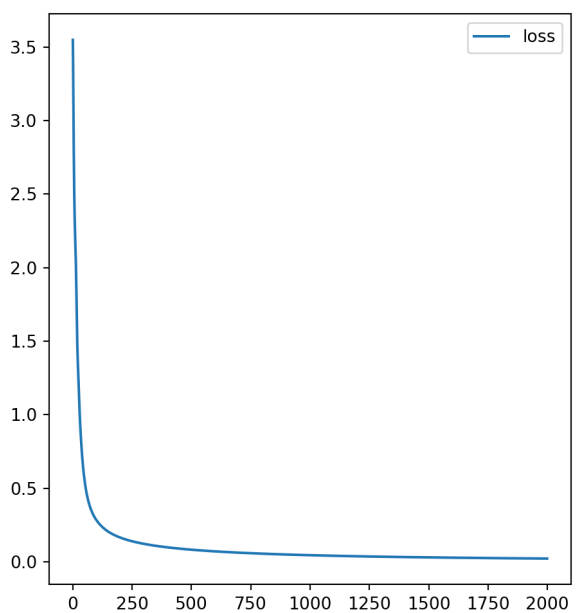
Performance - ReLU

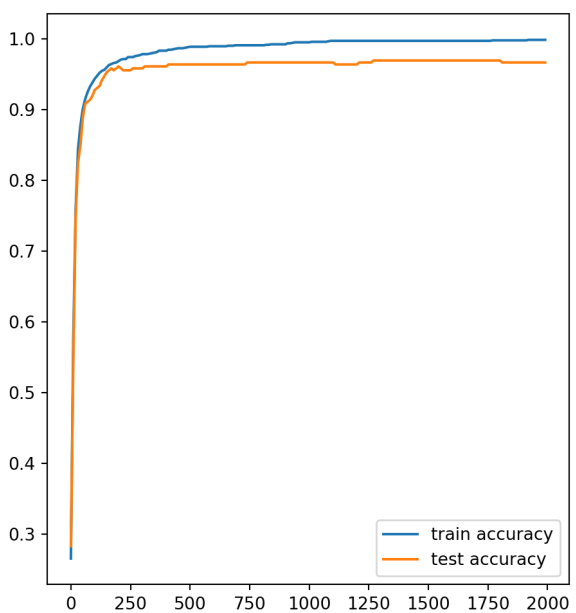
```
loss, train_acc, test_acc = mnist_fnn_model(64,64,10).fit(
  X_train, y_train, X_test, y_test, n=2000
)
train_acc[-5:]
```

```
[tensor(0.9986), tensor(0.9986), tensor(0.9986), tensor(0.9986), tensor(0.9986)]
```

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

```
[tensor(0.9667), tensor(0.9667), tensor(0.9667), tensor(0.9667), tensor(0.9667)]
```





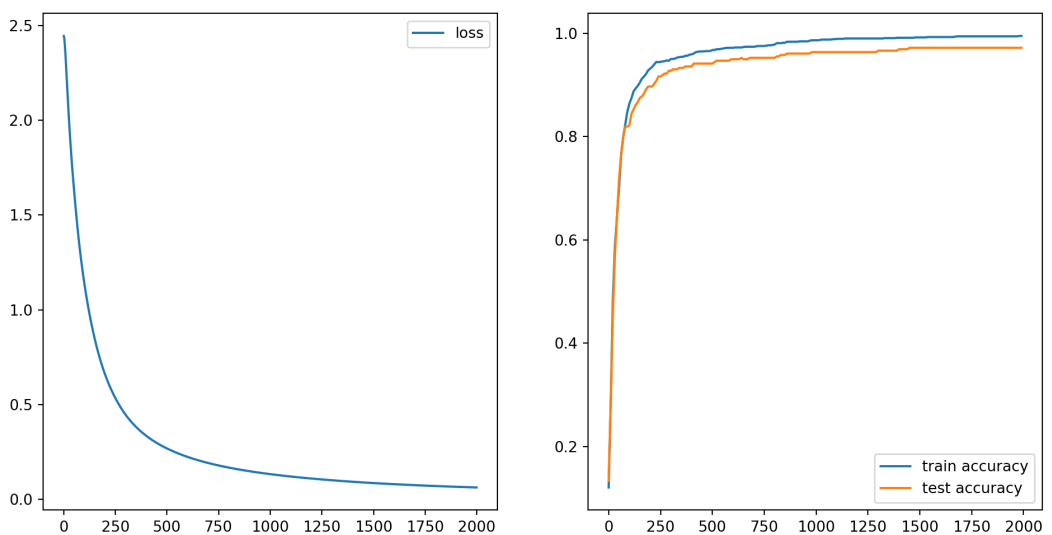
Performance - tanh

```
loss, train_acc, test_acc = mnist_fnn_model(
  64,64,10, nl_step=torch.nn.Tanh()
).fit(
  X_train, y_train, X_test, y_test, n=2000
)
train_acc[-5:]
```

```
[tensor(0.9944), tensor(0.9944), tensor(0.9944), tensor(0.9951), tensor(0.9951)]
```

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

```
[tensor(0.9722), tensor(0.9722), tensor(0.9722), tensor(0.9722), tensor(0.9722)]
```



Adding another layer

```
class mnist_fnn2_model(torch.nn.Module):
    def __init__(self, input_dim, hidden_dim, output_dim, nl_step = torch.nn.ReLU(), seed=1234):
        super().__init__()
        self.l1 = torch.nn.Linear(input_dim, hidden_dim)
        self.nl = nl_step
        self.l2 = torch.nn.Linear(hidden_dim, hidden_dim)
        self.nl = nl_step
        self.l3 = torch.nn.Linear(hidden_dim, output_dim)

    def forward(self, X):
        out = self.l1(X)
        out = self.nl(out)
        out = self.l2(out)
        out = self.nl(out)
        out = self.l3(out)
        return out

    def fit(self, X_train, y_train, X_test, y_test, lr=0.001, n=1000, acc_step=10):
      loss_fn = torch.nn.CrossEntropyLoss()
      opt = torch.optim.SGD(self.parameters(), lr=lr, momentum=0.9)
      losses, train_acc, test_acc = [], [], []

      for i in range(n):
```

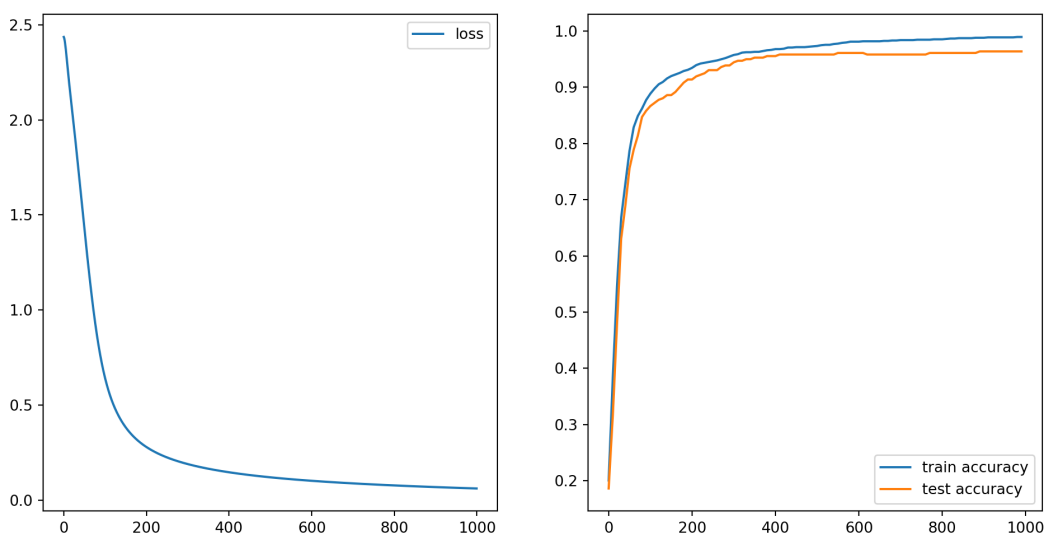
Performance - relu

```
loss, train_acc, test_acc = mnist_fnn2_model(
  64,64,10, nl_step=torch.nn.ReLU()
).fit(
  X_train, y_train, X_test, y_test, n=1000
)
train_acc[-5:]
```

```
[tensor(0.9889), tensor(0.9889), tensor(0.9889), tensor(0.9896), tensor(0.9896)]
```

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

```
[tensor(0.9639), tensor(0.9639), tensor(0.9639), tensor(0.9639), tensor(0.9639)]
```



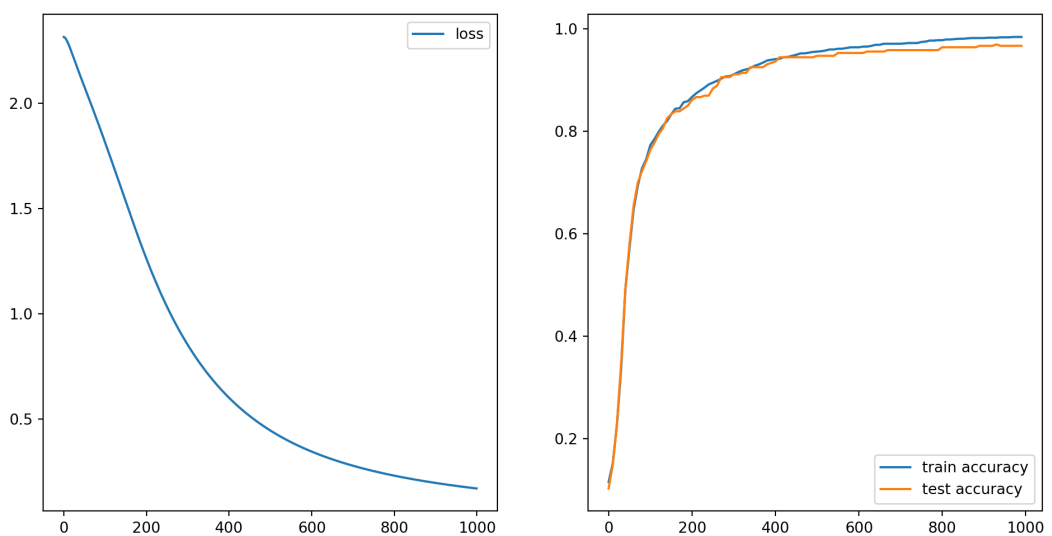
Performance - tanh

```
loss, train_acc, test_acc = mnist_fnn2_model(
  64,64,10, nl_step=torch.nn.Tanh()
).fit(
  X_train, y_train, X_test, y_test, n=1000
)
train_acc[-5:]
```

```
[tensor(0.9833), tensor(0.9833), tensor(0.9840), tensor(0.9840), tensor(0.9840)]
```

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

```
[tensor(0.9667), tensor(0.9667), tensor(0.9667), tensor(0.9667), tensor(0.9667)]
```

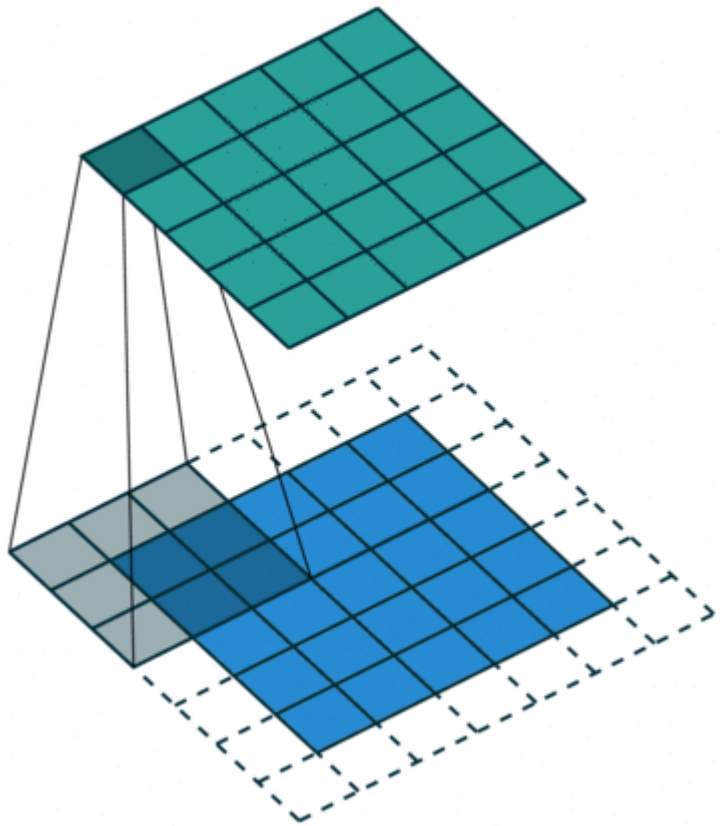


Convolutional NN

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



Animation source

nn.Conv2d()

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

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

```
Parameter containing:
tensor([[[[-0.2126, -0.0904,  0.1491],
          [-0.3284, -0.0247,  0.2515],
          [ 0.1292,  0.2992,  0.2095]]],


        [[[ 0.0066,  0.0926,  0.1636],
          [ 0.1279,  0.0875,  0.1772],
          [-0.2525, -0.3064, -0.1875]]],


        [[[ 0.1264, -0.2146,  0.3322],
          [ 0.2963, -0.2139,  0.0800],
          [-0.0684,  0.3225, -0.2510]]],


        [[[-0.3048, -0.1349, -0.2028],
          [-0.2928,  0.2831,  0.0100],
          [-0.3206, -0.1806,  0.1542]]]], requires_g
```

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

```
Parameter containing:
tensor([0.2514, 0.0478, 0.1583, 0.2862], requires_gra
```

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]])
```

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

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

```
tensor([[[  0.2514,   2.1372,   8.8115,   9.9783,   0.5591,  -2.5852,   0.2514,
            0.2514],
         [  0.2514,   5.6581,  12.7558,   4.9740,  -6.5905,  -4.0571,   0.2514,
            0.2514],
         [  0.2514,   8.2995,  11.0063,  -2.6590,  -6.6684,  -1.0240,   0.2514,
            0.2514],
         [  0.4609,   9.4512,  10.1341,  -1.7516,  -1.4324,   1.8837,   0.5099,
            0.2514],
         [  0.5029,   8.6012,   9.7387,   2.8518,   4.6978,   6.5248,   4.8637,
            1.8431],
```

```
 [  0.4005,   6.8824,   9.6466,   5.0654,   3.8233,   3.7298,   1.7526,
    0.2076],
 [  0.2514,   3.8012,   7.2894,   4.7997,   3.4224,   3.2588,  -4.1872,
   -6.6711],
 [  0.2514,   1.2950,   4.2678,   2.6568,  -2.7527,  -4.9629,  -7.2134,
   -5.1532]],

[[  0.0478,  -1.6394,  -3.9379,  -5.4284,  -3.5895,  -0.0607,   0.0478,
    0.0478],
 [  0.0478,  -1.1697,  -1.7273,  -1.3834,   0.4213,   0.8880,   0.0478,
    0.0478],
 [  0.0478,   1.5533,   0.9504,  -0.9727,  -0.1531,   0.0877,   0.0478,
    0.0478],
```

Pooling

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

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

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

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

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

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

```
p = torch.nn.AvgPool2d(kernel_size=2)
p(x)
```

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

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

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

Convolutional model

```
class mnist_conv_model(torch.nn.Module):
    def __init__(self):
        super().__init__()
        self.cnn  = torch.nn.Conv2d(
          in_channels=1, out_channels=8,
          kernel_size=3, stride=1, padding=1
        )
        self.relu = torch.nn.ReLU()
        self.pool = torch.nn.MaxPool2d(kernel_size=2)
        self.lin  = torch.nn.Linear(8 * 4 * 4, 10)

    def forward(self, X):
        out = self.cnn(X.view(-1, 1, 8, 8))
        out = self.relu(out)
        out = self.pool(out)
        out = self.lin(out.view(-1, 8 * 4 * 4))
        return out

    def fit(self, X_train, y_train, X_test, y_test, lr=0.001, n=1000, acc_step=10):
      loss_fn = torch.nn.CrossEntropyLoss()
      opt = torch.optim.SGD(self.parameters(), lr=lr, momentum=0.9)
      losses, train_acc, test_acc = [], [], []

```

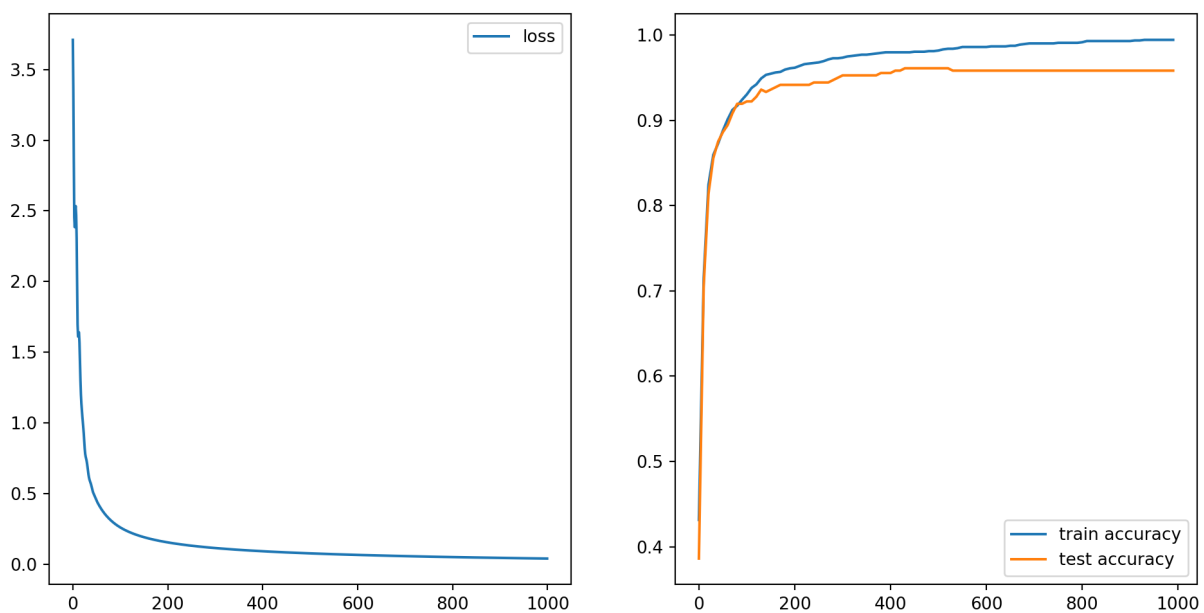
Performance

```
loss, train_acc, test_acc = mnist_conv_model().fit(
  X_train, y_train, X_test, y_test, n=1000
)
train_acc[-5:]
```

```
[tensor(0.9944), tensor(0.9944), tensor(0.9944), tensor(0.9944), tensor(0.9944)]
```

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

```
[tensor(0.9583), tensor(0.9583), tensor(0.9583), tensor(0.9583), tensor(0.9583)]
```



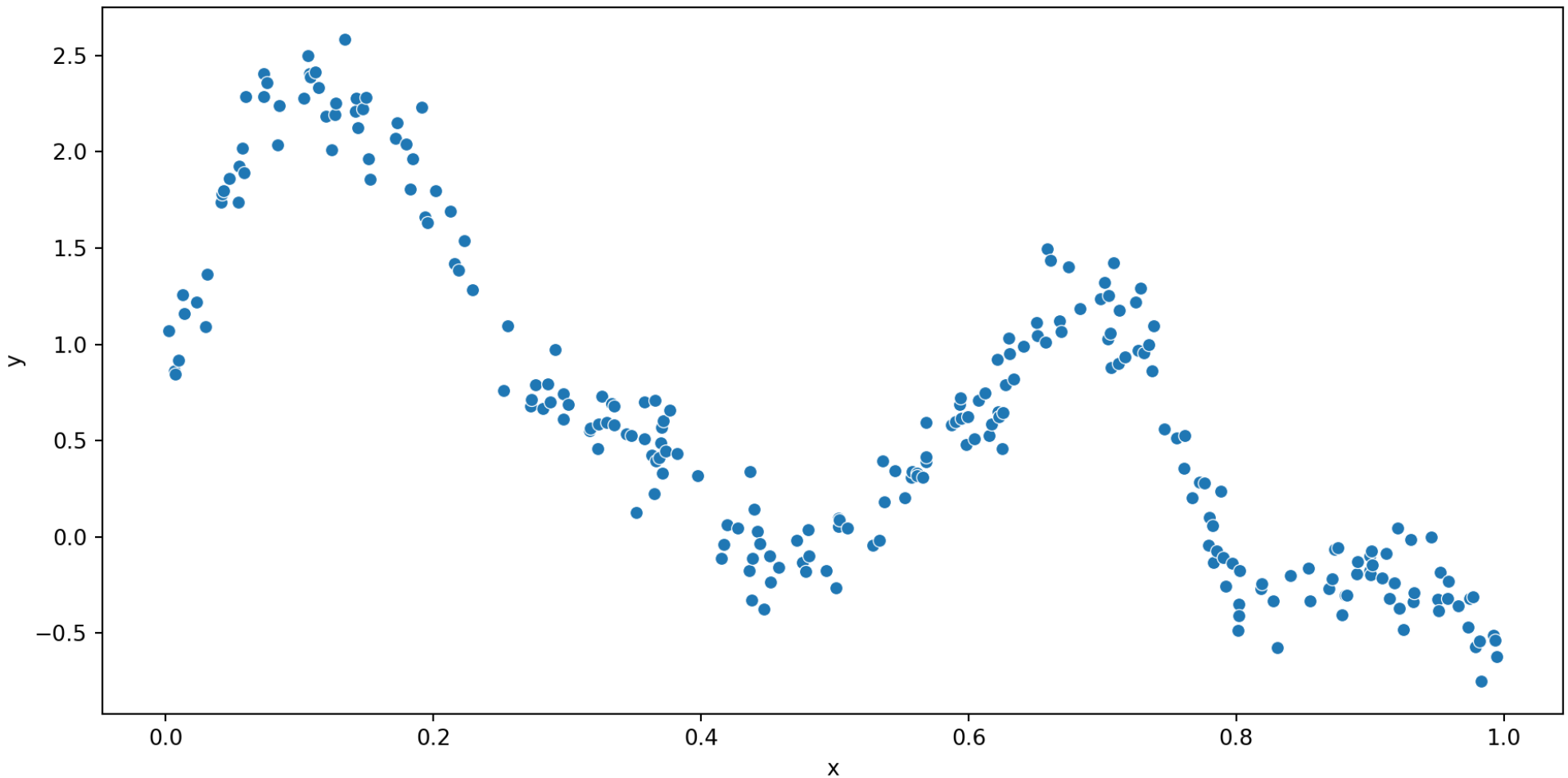
Cleaning up models

```
class mnist_conv_model2(torch.nn.Module):
    def __init__(self):
        super().__init__()
        self.model = torch.nn.Sequential(
          torch.nn.Unflatten(1, (1,8,8)),
          torch.nn.Conv2d(
            in_channels=1, out_channels=8,
            kernel_size=3, stride=1, padding=1
          ),
          torch.nn.ReLU(),
          torch.nn.MaxPool2d(kernel_size=2),
          torch.nn.Flatten(),
          torch.nn.Linear(8 * 4 * 4, 10)
        )

    def forward(self, X):
        return self.model(X)

    def fit(self, X_train, y_train, X_test, y_test, lr=0.001, n=1000, acc_step=10):
      opt = torch.optim.SGD(self.parameters(), lr=lr, momentum=0.9)
      losses, train_acc, test_acc = [], [], []

      for i in range(n):
```

A bit more on non-linear activation layers

Non-linear functions

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



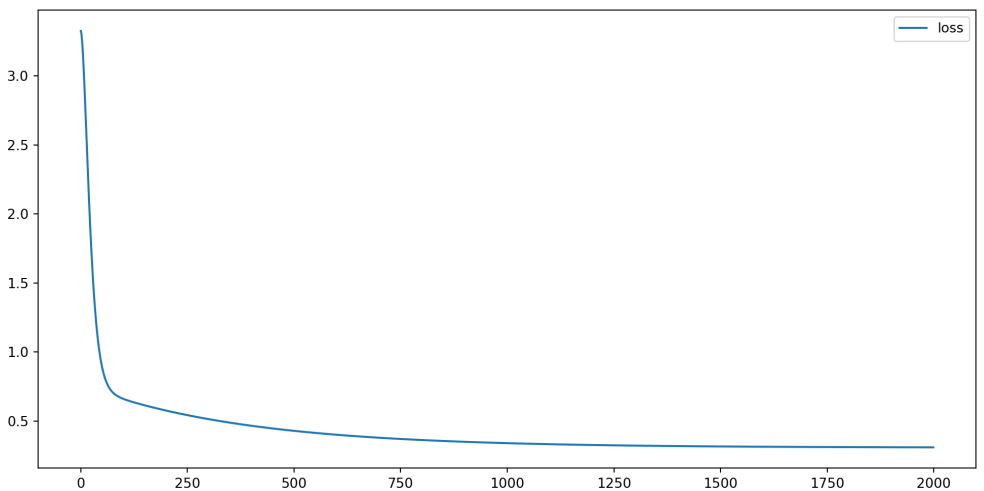
Linear regression

```
class lin_reg(torch.nn.Module):
    def __init__(self, X):
        super().__init__()
        self.n = X.shape[0]
        self.p = X.shape[1]
        self.model = torch.nn.Sequential(
          torch.nn.Linear(self.p, self.p)
        )

    def forward(self, X):
        return self.model(X)

    def fit(self, X, y, n=1000):
      losses = []
      opt = torch.optim.SGD(self.parameters(), lr=0.001, momentum=0.9)
      for i in range(n):
          loss = torch.nn.MSELoss()(self(X).squeeze(), y)
          loss.backward()
          opt.step()
          opt.zero_grad()
          losses.append(loss.item())

      return losses
```

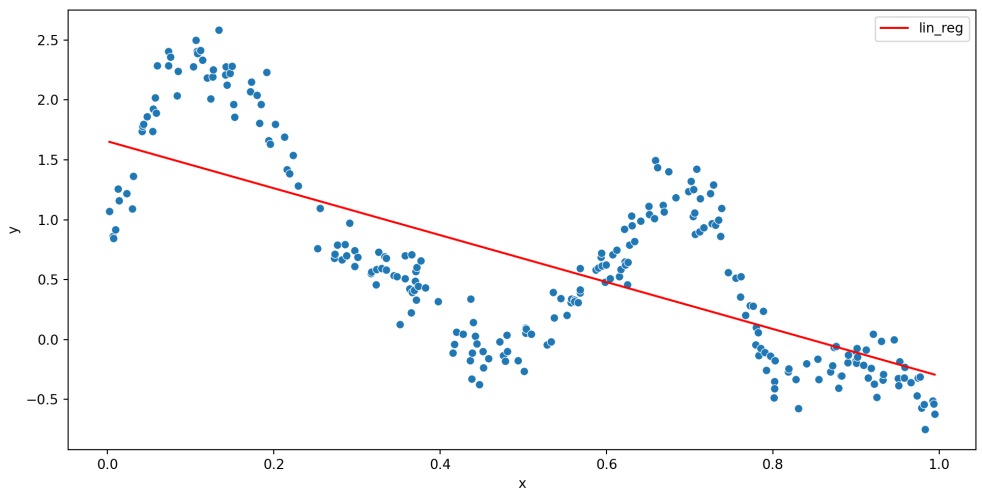
Model results

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

Training loss:



Predictions



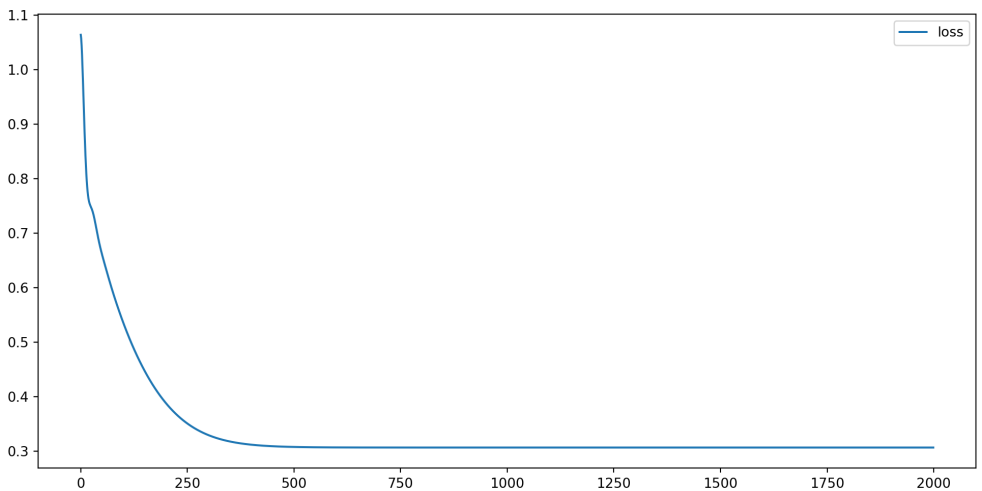
Double linear regression

```
class dbl_lin_reg(torch.nn.Module):
    def __init__(self, X, hidden_dim=10):
        super().__init__()
        self.n = X.shape[0]
        self.p = X.shape[1]
        self.model = torch.nn.Sequential(
          torch.nn.Linear(self.p, hidden_dim),
          torch.nn.Linear(hidden_dim, 1)
        )

    def forward(self, X):
        return self.model(X)

    def fit(self, X, y, n=1000):
      losses = []
      opt = torch.optim.SGD(self.parameters(), lr=0.001, momentum=0.9)
      for i in range(n):
          loss = torch.nn.MSELoss()(self(X).squeeze(), y)
          loss.backward()
          opt.step()
          opt.zero_grad()
          losses.append(loss.item())

```

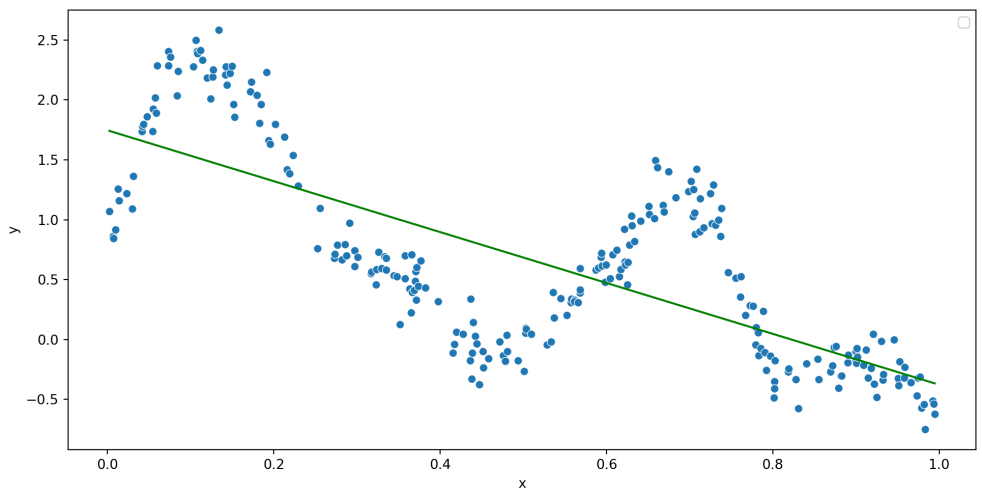
Model results

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

Training loss:



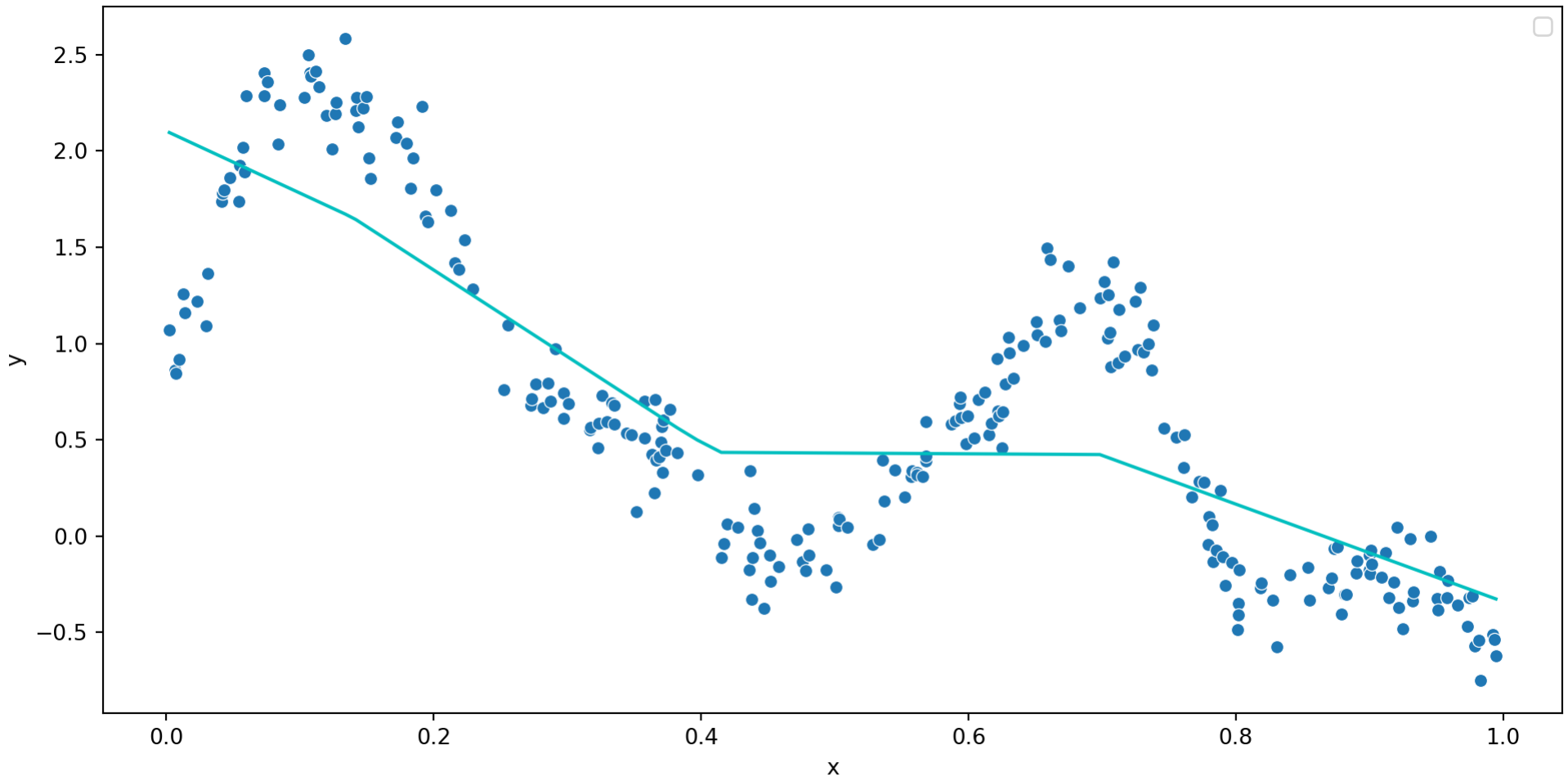
Predictions



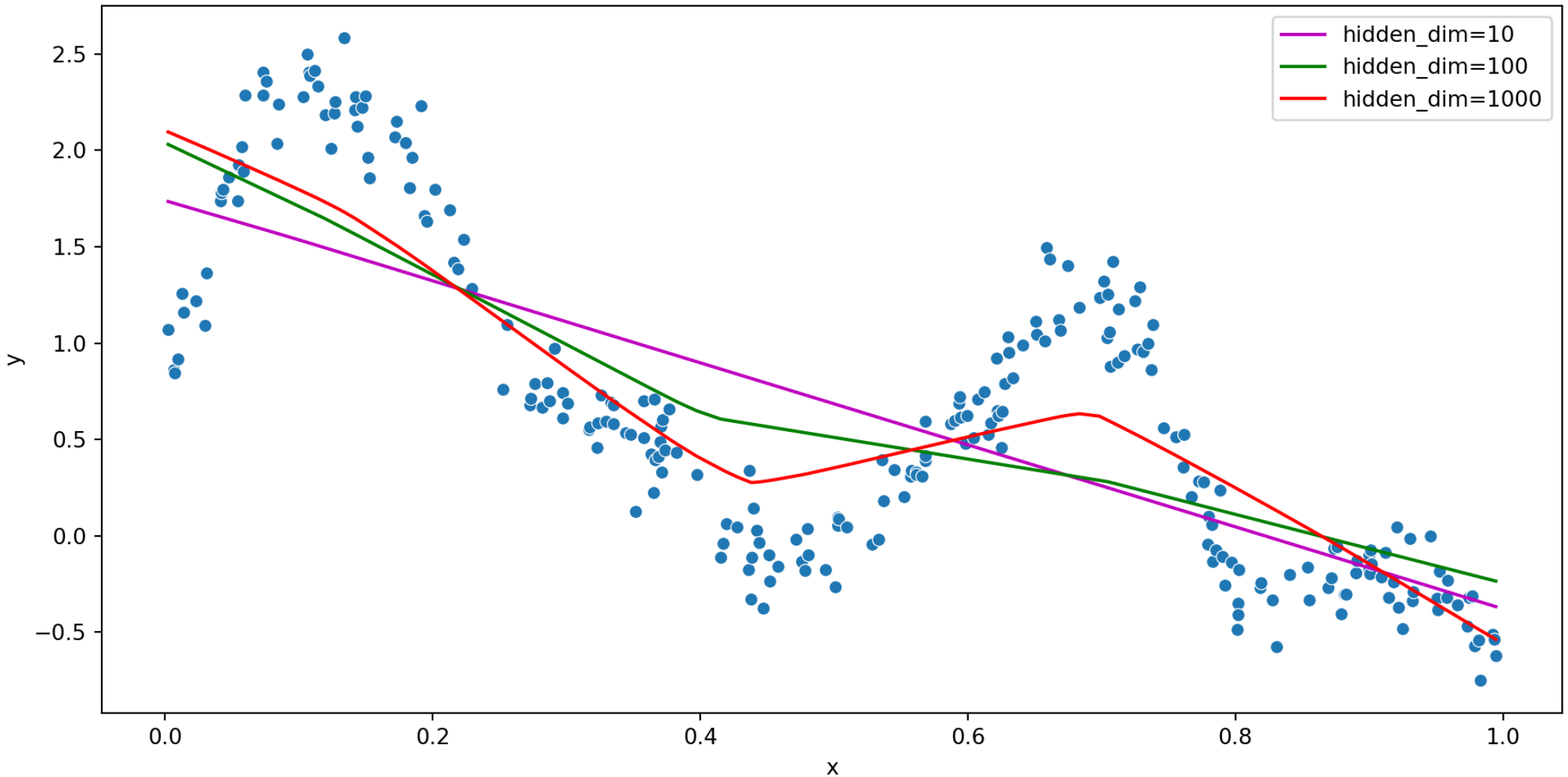
Non-linear regression w/ ReLU

```
class lin_reg_relu(torch.nn.Module):
    def __init__(self, X, hidden_dim=100):
        super().__init__()
        self.n = X.shape[0]
        self.p = X.shape[1]
        self.model = torch.nn.Sequential(
          torch.nn.Linear(self.p, hidden_dim),
          torch.nn.ReLU(),
          torch.nn.Linear(hidden_dim, 1)
        )

    def forward(self, X):
        return self.model(X)

    def fit(self, X, y, n=1000):
      losses = []
      opt = torch.optim.SGD(self.parameters(), lr=0.001, momentum=0.9)
      for i in range(n):
          loss = torch.nn.MSELoss()(self(X).squeeze(), y)
          loss.backward()
          opt.step()
          opt.zero_grad()
          losses.append(loss.item())
```

Model results



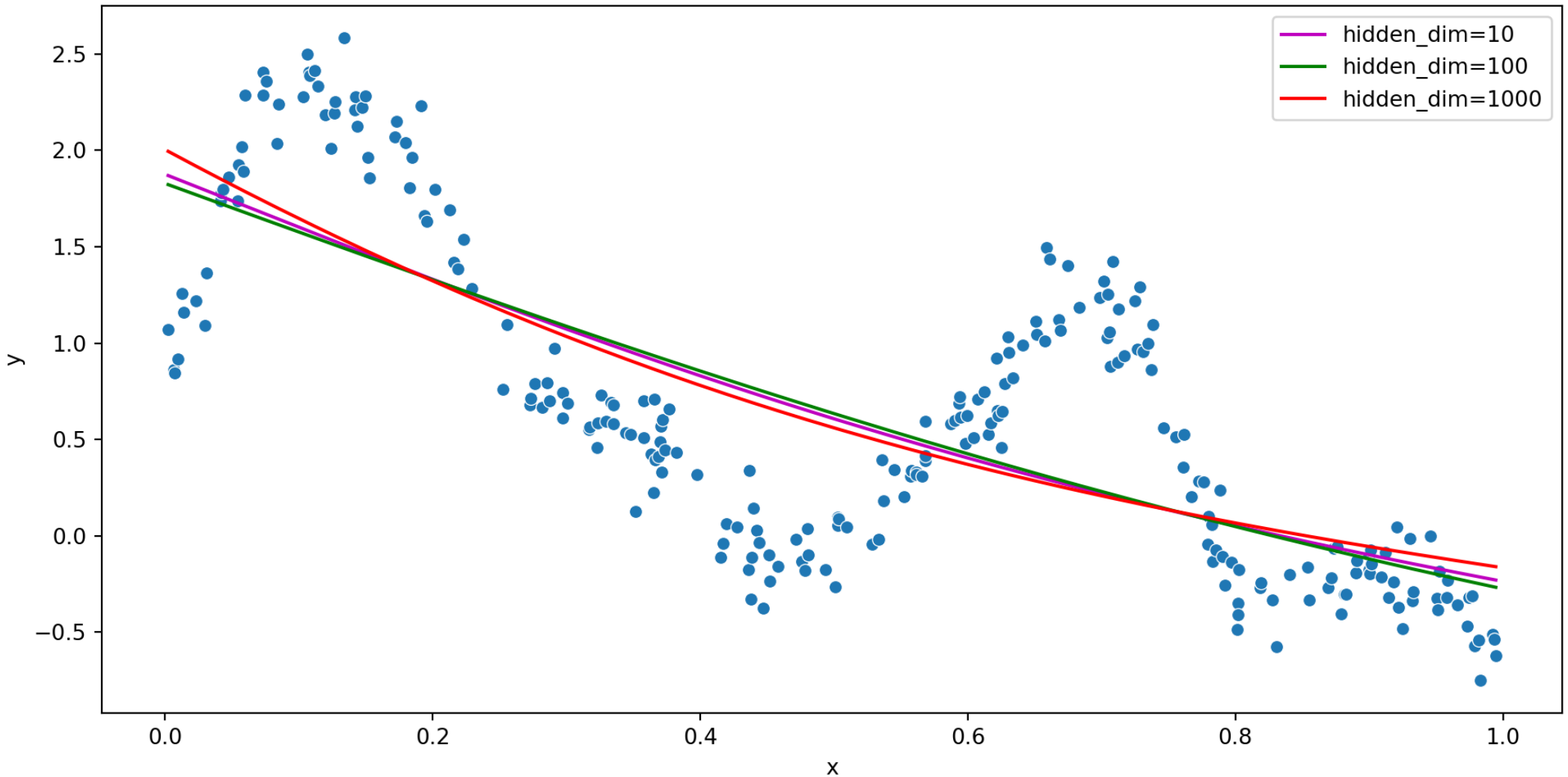
Hidden dimensions



Non-linear regression w/ Tanh

```
class lin_reg_tanh(torch.nn.Module):
    def __init__(self, X, hidden_dim=10):
        super().__init__()
        self.n = X.shape[0]
        self.p = X.shape[1]
        self.model = torch.nn.Sequential(
          torch.nn.Linear(self.p, hidden_dim),
          torch.nn.Tanh(),
          torch.nn.Linear(hidden_dim, 1)
        )

    def forward(self, X):
        return self.model(X)

    def fit(self, X, y, n=1000):
      losses = []
      opt = torch.optim.SGD(self.parameters(), lr=0.001, momentum=0.9)
      for i in range(n):
          loss = torch.nn.MSELoss()(self(X).squeeze(), y)
          loss.backward()
          opt.step()
          opt.zero_grad()
          losses.append(loss.item())
```

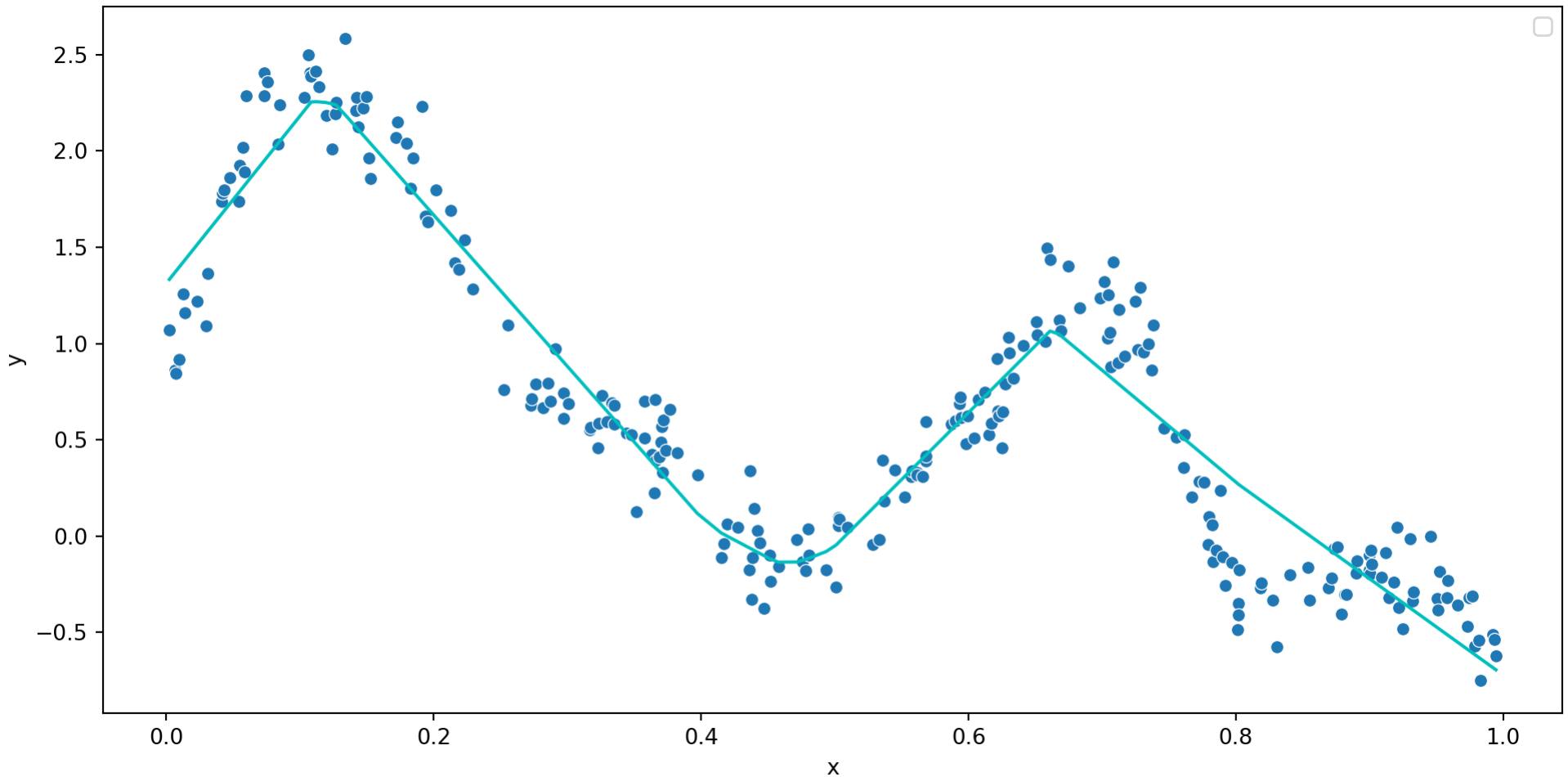
Tanh & hidden dimension



Three layers

```
class three_layers(torch.nn.Module):
    def __init__(self, X, hidden_dim=100):
        super().__init__()
        self.n = X.shape[0]
        self.p = X.shape[1]
        self.model = torch.nn.Sequential(
          torch.nn.Linear(self.p, hidden_dim),
          torch.nn.ReLU(),
          torch.nn.Linear(hidden_dim, hidden_dim),
          torch.nn.ReLU(),
          torch.nn.Linear(hidden_dim, 1)
        )

    def forward(self, X):
        return self.model(X)

    def fit(self, X, y, n=1000):
      losses = []
      opt = torch.optim.SGD(self.parameters(), lr=0.001, momentum=0.9)
      for i in range(n):
          loss = torch.nn.MSELoss()(self(X).squeeze(), y)
          loss.backward()
          opt.step()
```

Model results



Five layers

```
class five_layers(torch.nn.Module):
    def __init__(self, X, hidden_dim=100):
        super().__init__()
        self.n = X.shape[0]
        self.p = X.shape[1]
        self.model = torch.nn.Sequential(
          torch.nn.Linear(self.p, hidden_dim),
          torch.nn.ReLU(),
          torch.nn.Linear(hidden_dim, hidden_dim),
          torch.nn.ReLU(),
          torch.nn.Linear(hidden_dim, hidden_dim),
          torch.nn.ReLU(),
          torch.nn.Linear(hidden_dim, hidden_dim),
          torch.nn.ReLU(),
          torch.nn.Linear(hidden_dim, 1)
        )

    def forward(self, X):
        return self.model(X)

    def fit(self, X, y, n=1000):
      losses = []
      opt = torch.optim.SGD(self.parameters(), lr=0.001, momentum=0.9)
```

Model results