pytorch - nn

Lecture 18

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 proceedure:

- 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

```
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()
```

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

```
class greet:
  def __init__(self, greeting):
    self.greeting = greeting
  def __call__(self, name):
    return self.greeting + " " + name
```

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

```
'Hello Jane'
```

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

```
'Good morning Bob'
```

MNIST & Logistic models

MNIST handwritten digits - simplified

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

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

```
(1797, 64)
```

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

```
(1797,)
```

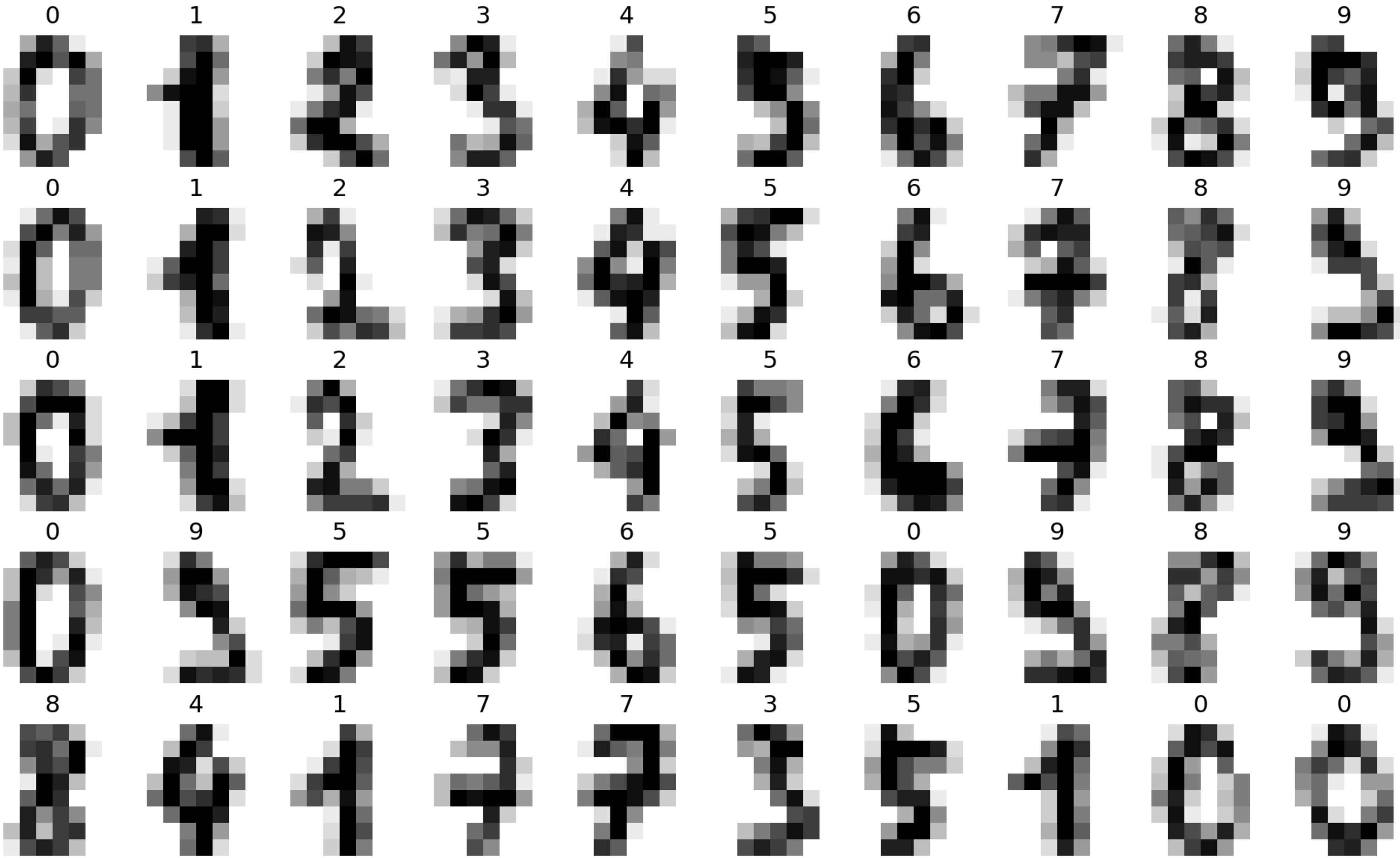
```python
X[0:2]
```

```
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.,
```

```python
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 LogisticReg
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 Torch 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])
```

```
X_train.dtype
```

```
torch.float32
```

```
y_train.shape
```

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

```
y_train.dtype
```

```
torch.int64
```

```
X_test.shape
```

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

```
X_test.dtype
```

```
torch.float32
```

```
y_test.shape
```

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

```
y_test.dtype
```

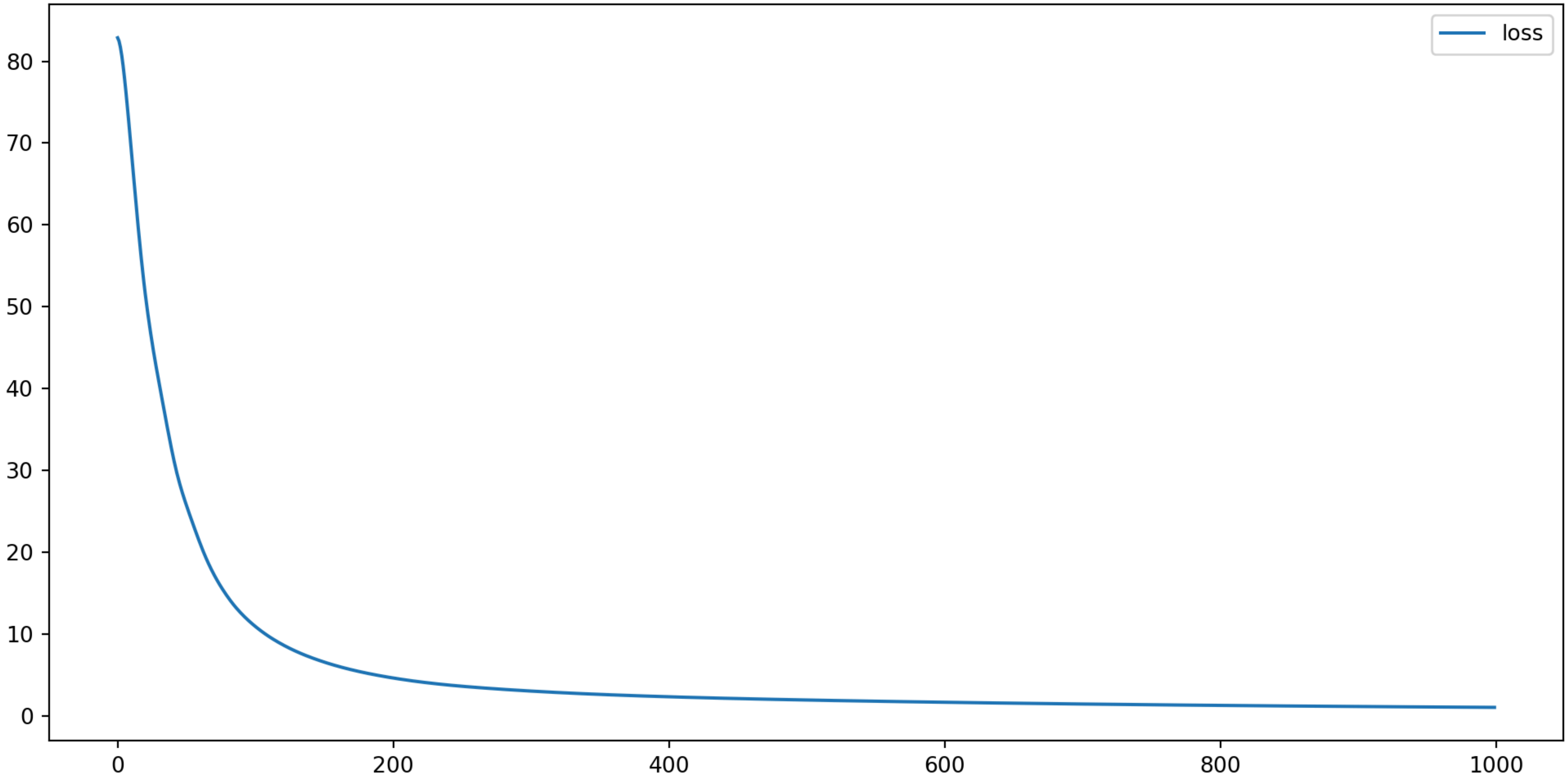
```
torch.int64
```

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()
```

Cross entropy loss

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



Cross entropy loss

From the pytorch documentation:

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

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

Out-of-sample accuracy

```
model(X_test)
```

```
tensor([[ -49.9200,  -67.9009,  -67.7915,
          -25.4841,    5.6110,  -25.7693,
          -51.8763,   25.9151,  -16.4802,
          -62.9280],
        [ -37.2071,   60.9700,   -1.3581,
           -5.8147,  -32.9665,  -43.7874,
          -36.7363,  -27.0210,  -26.2828,
           58.2118],
        [ -79.5292,  -45.3445,  -49.9198,
          -25.6674,  -21.3866,  -78.1699,
          -54.0964,   44.1203,  -19.2856,
          -51.4893],
        [  28.0738,  -38.6156,  -29.9736,
          -16.1450,    3.7312,   27.9189,
           43.4791,  -55.5872,    4.6998,
          -43.6908],
        [  33.7138,  -45.8625,  -32.5098,
```

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

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

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

```
tensor(333)
```

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

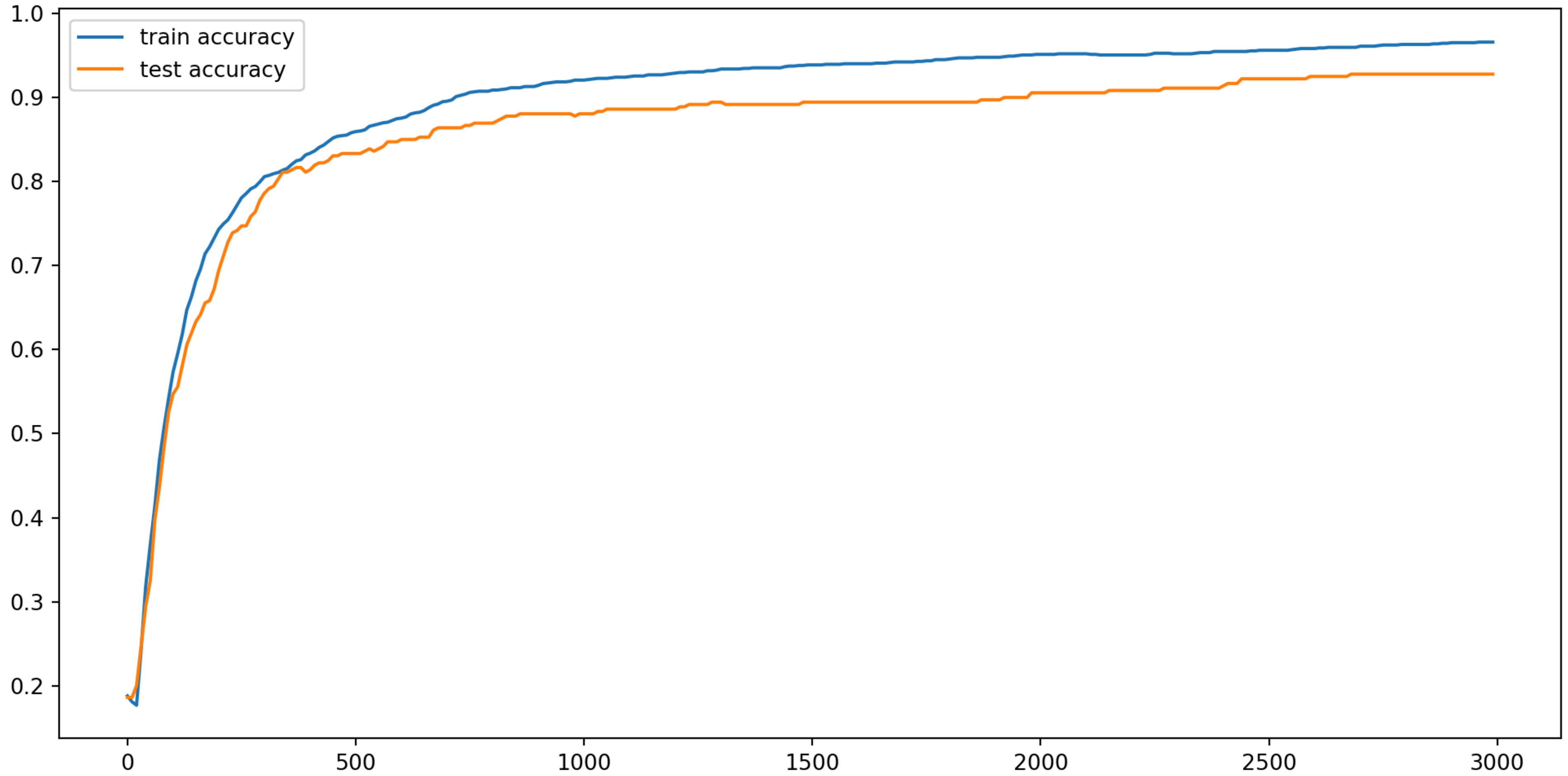
```
tensor(0.9250)
```

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()
```

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)
```

NN linear layer

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

$$y = XA^T + b$$

```
X.shape
```

```
(1797, 64)
```

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

```
<generator object Module.parameters at 0x17bf99c40>
```

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

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

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

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

Performance

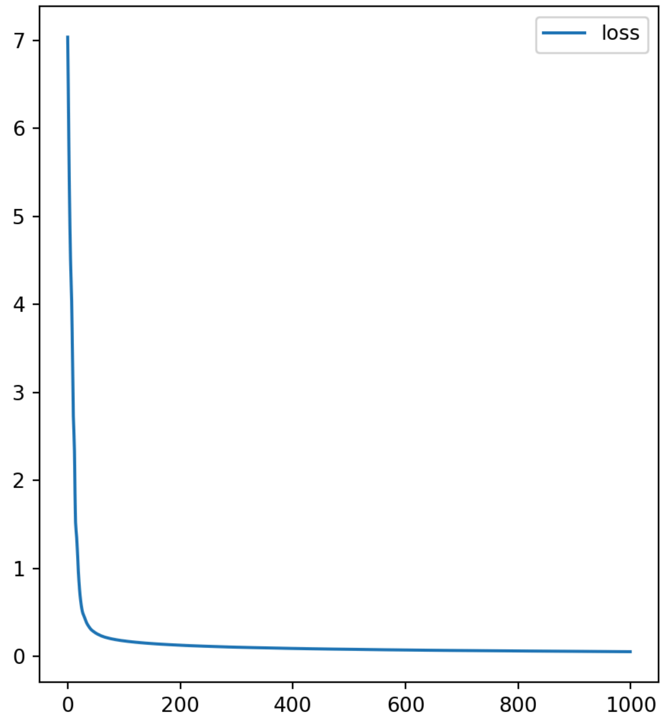
```
loss, train_acc, test_acc = model.fit(X_train, y_train, X_test, y_test, n=1000)
```

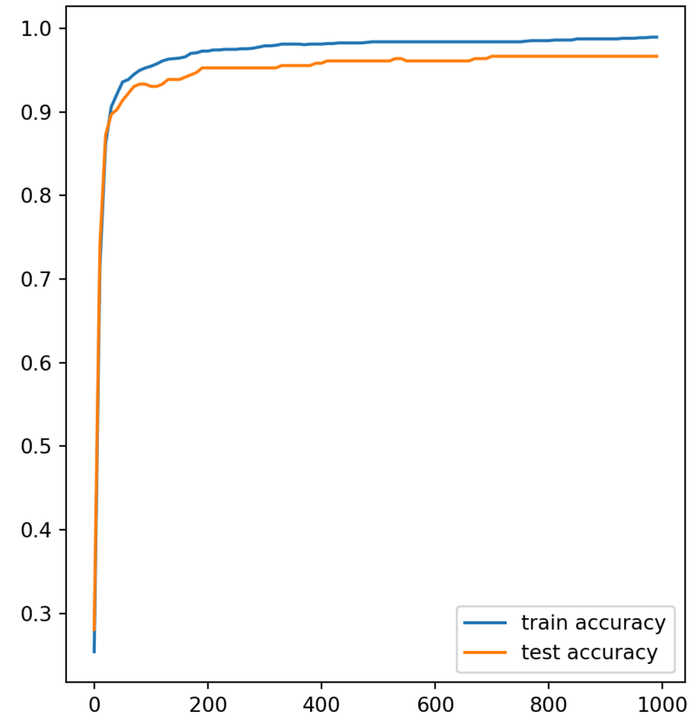
```
train_acc[-5:]
```

```
[tensor(0.9882), tensor(0.9889),
tensor(0.9889), tensor(0.9896), tensor(0.9896)]
```

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

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





Feedforward Neural Network

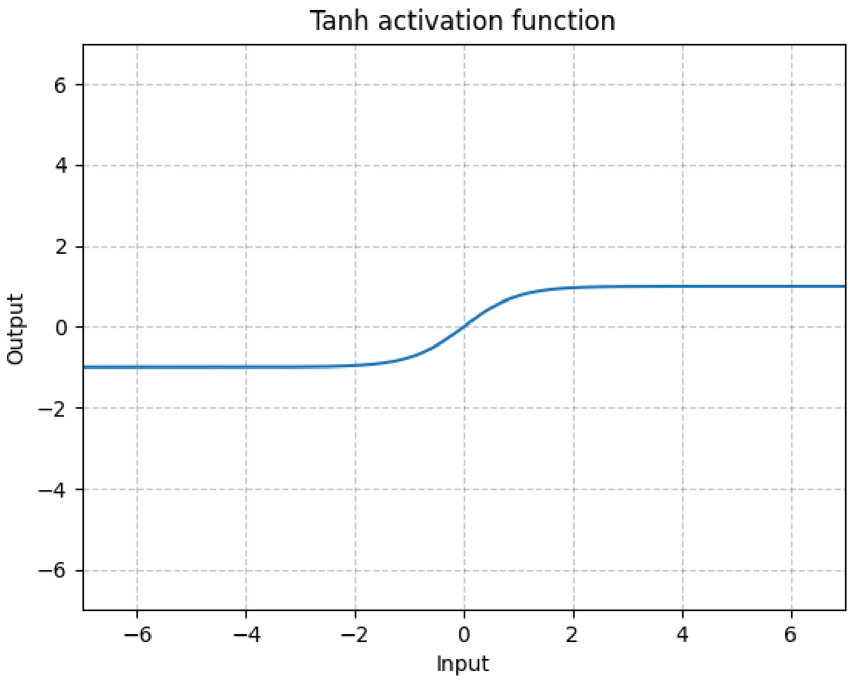
FNN Model

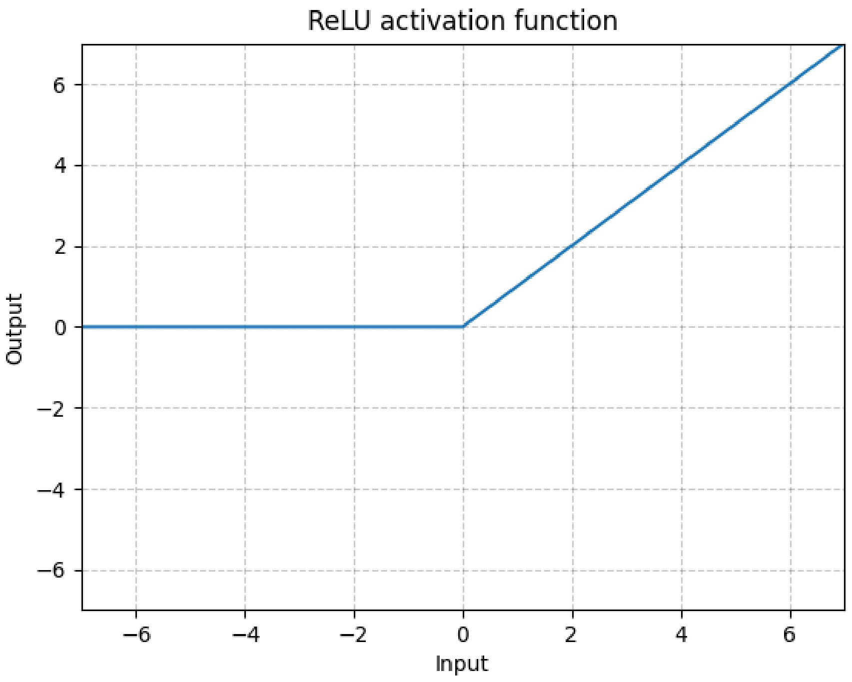
```
class mnist_fnn_model(torch.nn.Module):
    def __init__(self, input_dim, hidden_dim, output_dim, nl_step = torch.nn.ReLU(
        super().__init__()
        self.l1 = torch.nn.Linear(input_dim, hidden_dim)
        self.nl = nl_step
        self.l2 = torch.nn.Linear(hidden_dim, output_dim)

    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()
```

Non-linear activation functions

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

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





Model parameters

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

```
4
```

```python
for i, p in enumerate(model.parameters()):
  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

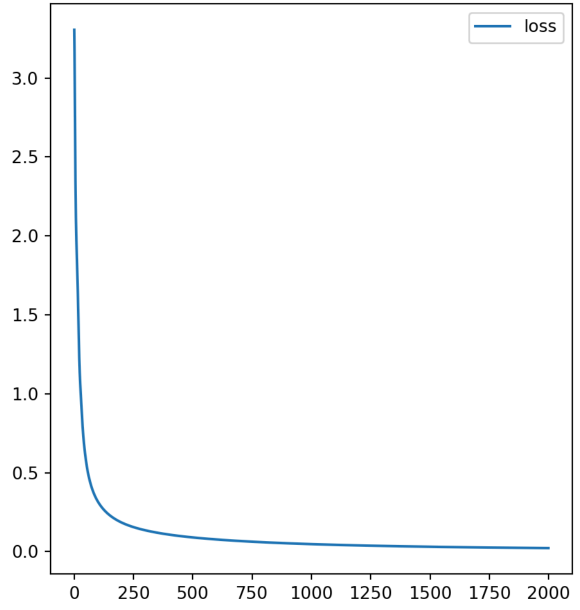
```
loss, train_acc, test_acc = mnist_fnn_model(64,64,10).fit(
  X_train, y_train, X_test, y_test, n=2000
)
```

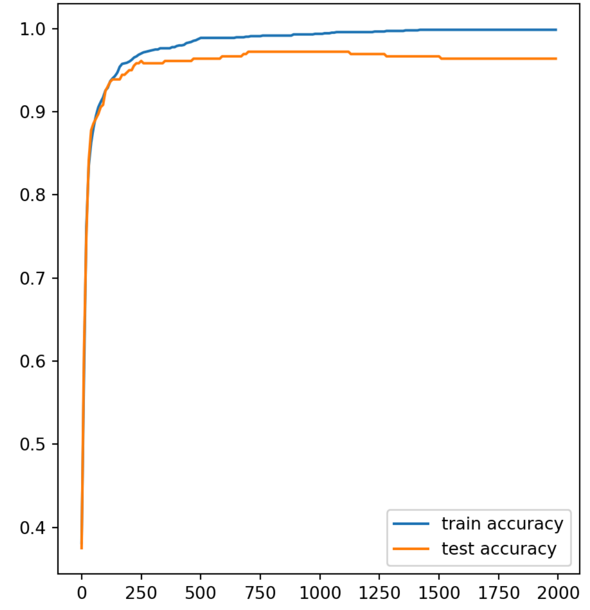
```
train_acc[-5:]
```

```
[0.9986082115518441, 0.9986082115518441,
0.9986082115518441, 0.9986082115518441,
0.9986082115518441]
```

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

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





Performance - tanh

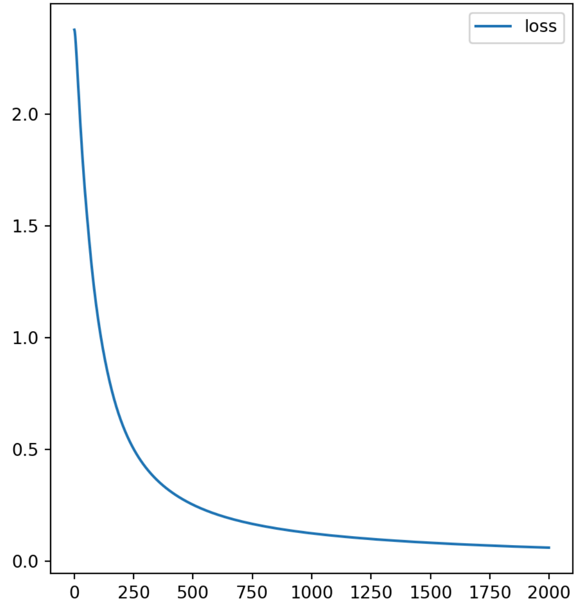
```
loss, train_acc, test_acc = mnist_fnn_model(64,64,10, nl_step=torch.nn.Tanh()).fit(
  X_train, y_train, X_test, y_test, n=2000
)
```

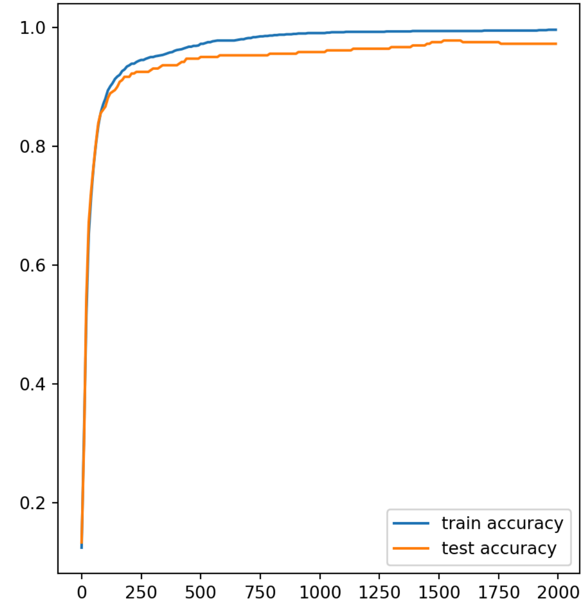
```
train_acc[-5:]
```

```
[0.9951287404314544, 0.9958246346555324,
0.9958246346555324, 0.9958246346555324,
0.9958246346555324]
```

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

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





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=123
        super().__init__()
        self.l1 = torch.nn.Linear(input_dim, hidden_dim)
        self.nl1 = nl_step
        self.l2 = torch.nn.Linear(hidden_dim, hidden_dim)
        self.nl2 = nl_step
        self.l3 = torch.nn.Linear(hidden_dim, output_dim)

    def forward(self, X):
        out = self.l1(X)
        out = self.nl1(out)
        out = self.l2(out)
        out = self.nl2(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):
          opt.zero_grad()
```

Performance - relu

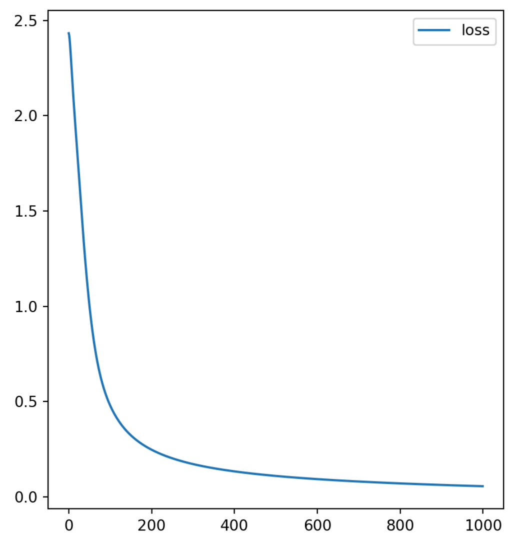
```python
loss, train_acc, test_acc = mnist_fnn2_model(
  64,64,10, nl_step=torch.nn.ReLU()
).fit(
  X_train, y_train, X_test, y_test, n=1000
)
```

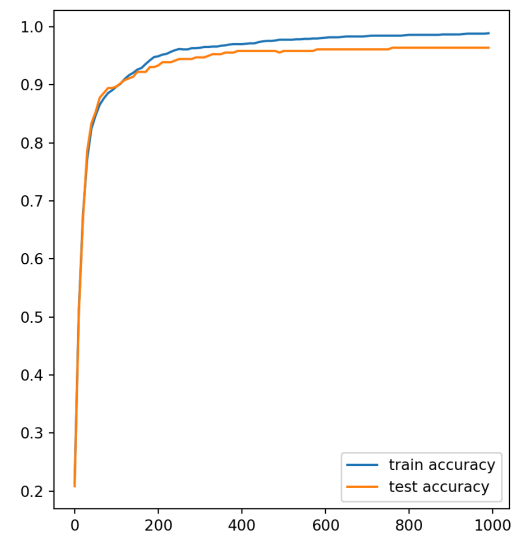
```python
train_acc[-5:]
```

```
[0.988169798190675, 0.988169798190675,
0.988169798190675, 0.988169798190675,
0.988865692414753]
```

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

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





Performance - tanh

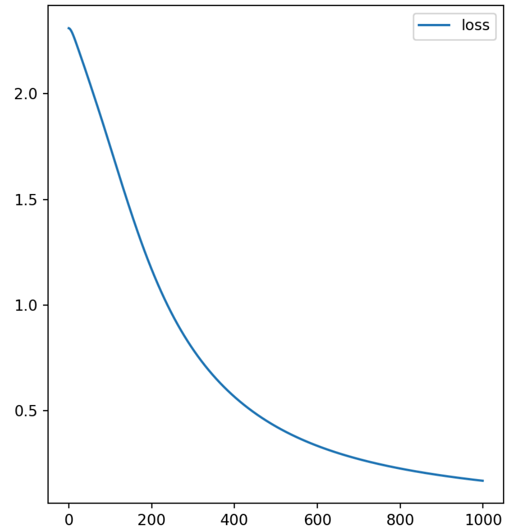
```
loss, train_acc, test_acc = mnist_fnn2_model(
  64,64,10, nl_step=torch.nn.Tanh()
).fit(
  X_train, y_train, X_test, y_test, n=1000
)
```

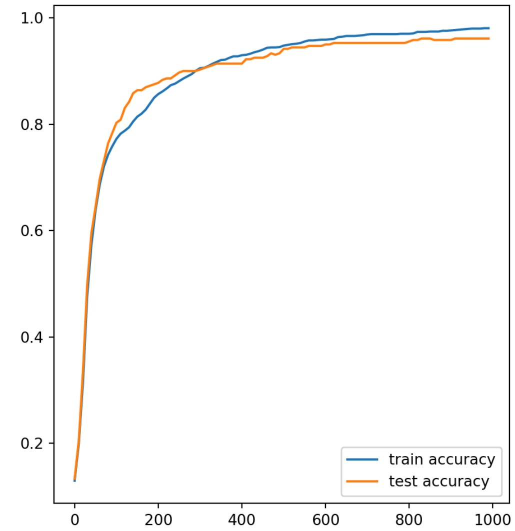
```
train_acc[-5:]
```

```
[0.9798190675017397, 0.9798190675017397,
0.9798190675017397, 0.9805149617258176,
0.9805149617258176]
```

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

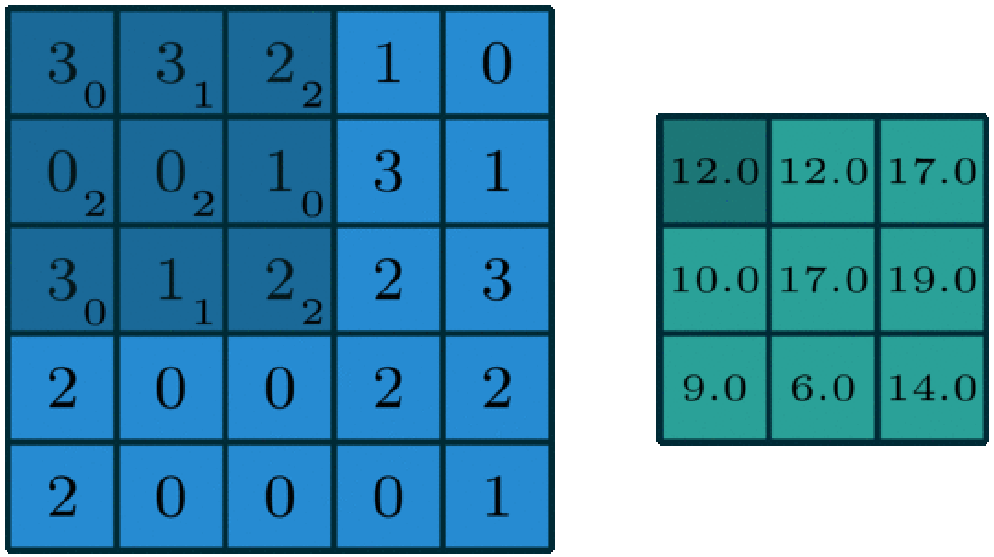
```
[0.9611111111111111, 0.9611111111111111,
0.9611111111111111, 0.9611111111111111,
0.9611111111111111]
```

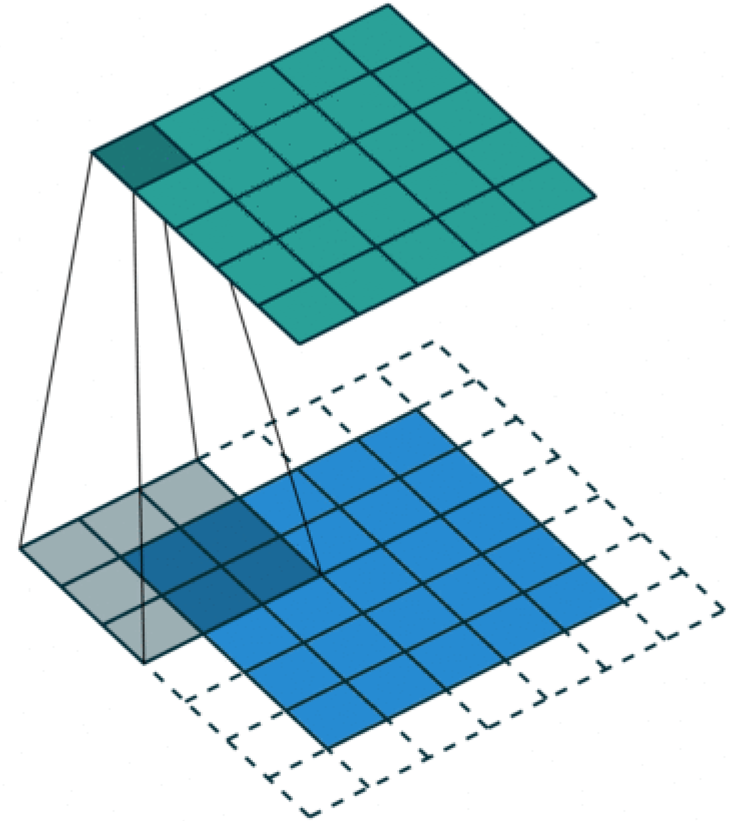




Convolutional NN

2d convolutions





nn.Conv2d()

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

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

```
Parameter containing:
tensor([[[[-0.0916, -0.1771,  0.2714],
          [ 0.2181, -0.3278, -0.1257],
          [-0.2952, -0.3100, -0.0557]]],


        [[[-0.2708, -0.1664,  0.2959],
          [-0.2933, -0.0739,  0.2642],
          [ 0.3022, -0.2897, -0.1198]]],


        [[[-0.1436, -0.2614, -0.3112],
          [-0.1443,  0.0939,  0.2256],
          [ 0.1558, -0.2530,  0.1756]]],


        [[[ 0.2582, -0.1870,  0.0828],
          [ 0.1618,  0.1541,  0.1748],
          [ 0.0206, -0.2393,  0.0881]]]],
       requires_grad=True)
```

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

```
Parameter containing:
tensor([-0.1378, -0.0058,  0.1867, -0.2046],
       requires_grad=True)
```

Applying `Conv2d()`

```
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.]])
```

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

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

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

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

```
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.]]])
```

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

```
tensor([[[-1.3784e-01, -6.3952e-01,
          -5.0761e+00, -1.2749e+01,
          -8.1465e+00,  4.8966e-01,
          -1.3784e-01, -1.3784e-01],
         [-1.3784e-01, -2.1049e+00,
          -7.7594e+00, -1.1418e+01,
          -5.3177e+00,  1.6247e-01,
          -1.3784e-01, -1.3784e-01],
         [-1.3784e-01, -3.6071e-01,
          -8.8376e+00, -1.0391e+01,
          -2.7838e+00, -6.8770e-01,
          -1.3784e-01, -1.3784e-01],
         [-1.9358e-01,  1.0275e+00,
```

Pooling

```python
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])
```

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

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

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

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

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

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

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

Performance

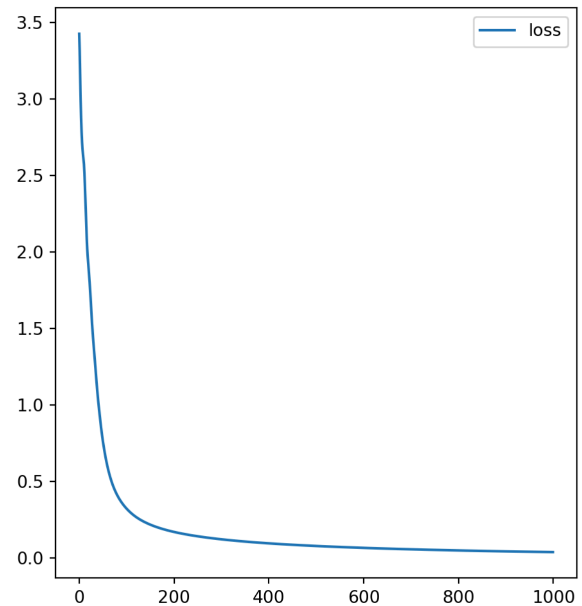
```
loss, train_acc, test_acc = mnist_conv_model().fit(
  X_train, y_train, X_test, y_test, n=1000
)
```

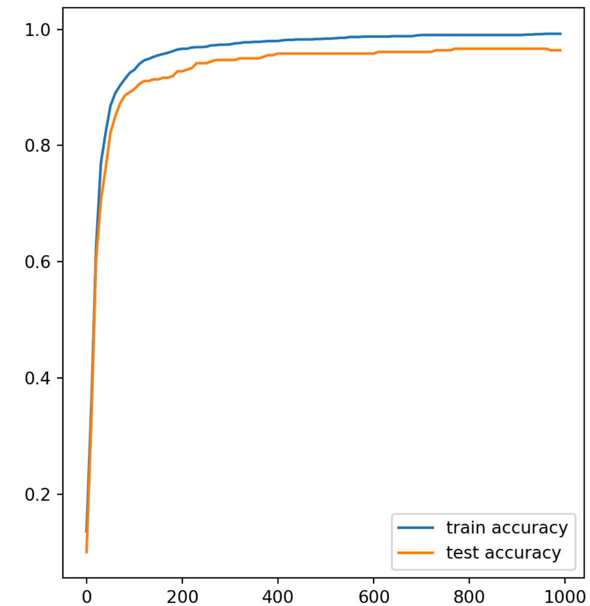
```
train_acc[-5:]
```

```
[0.9916492693110647, 0.9923451635351427,
0.9923451635351427, 0.9923451635351427,
0.9923451635351427]
```

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

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





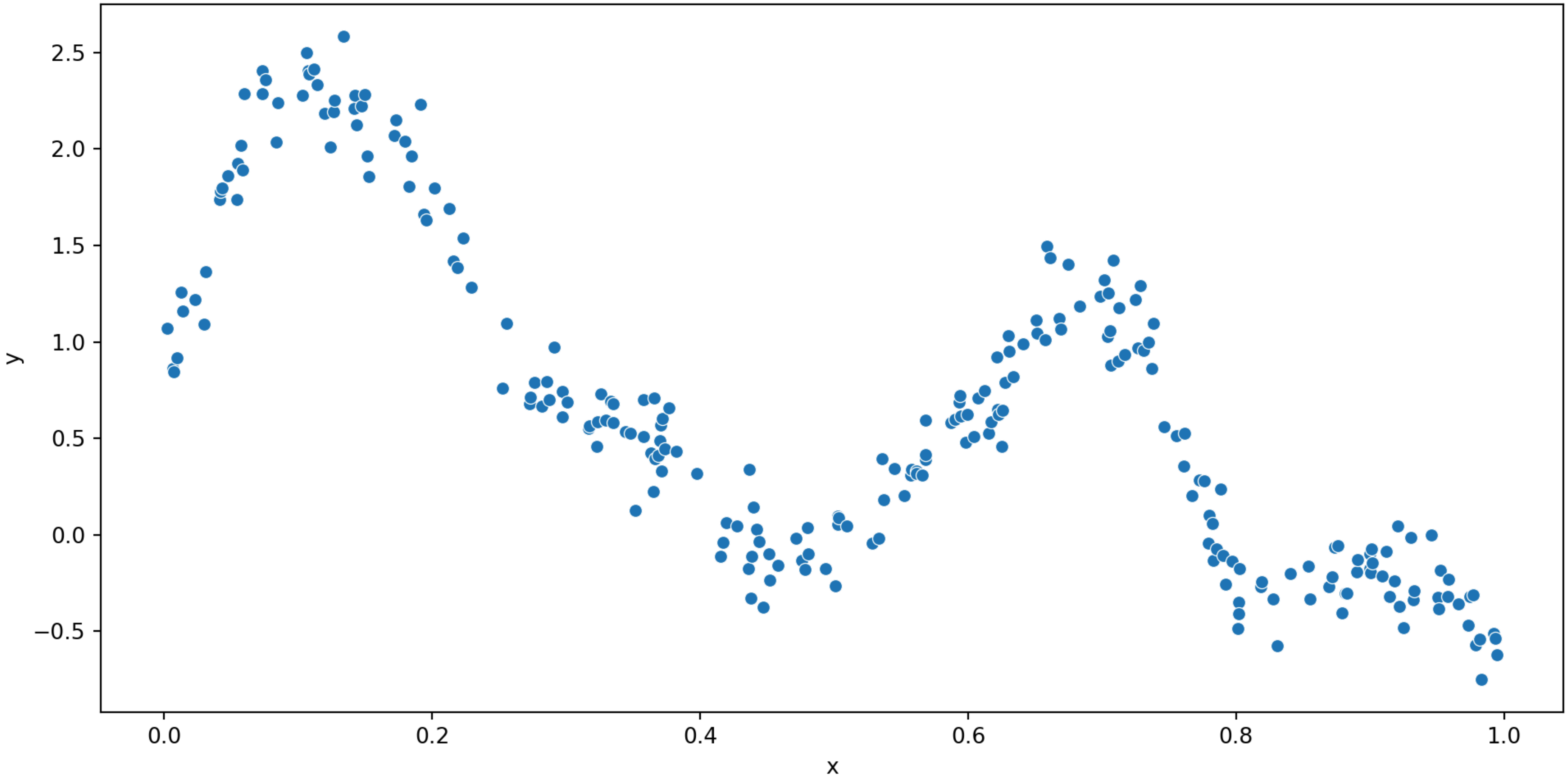
Organizing models

```
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):
          opt.zero_grad()
```

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)
```



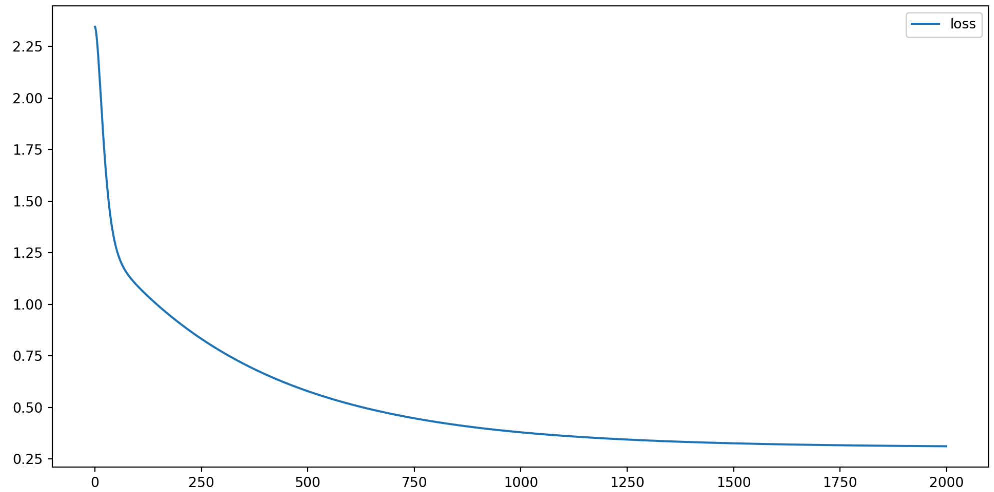
Linear regression

```
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
```

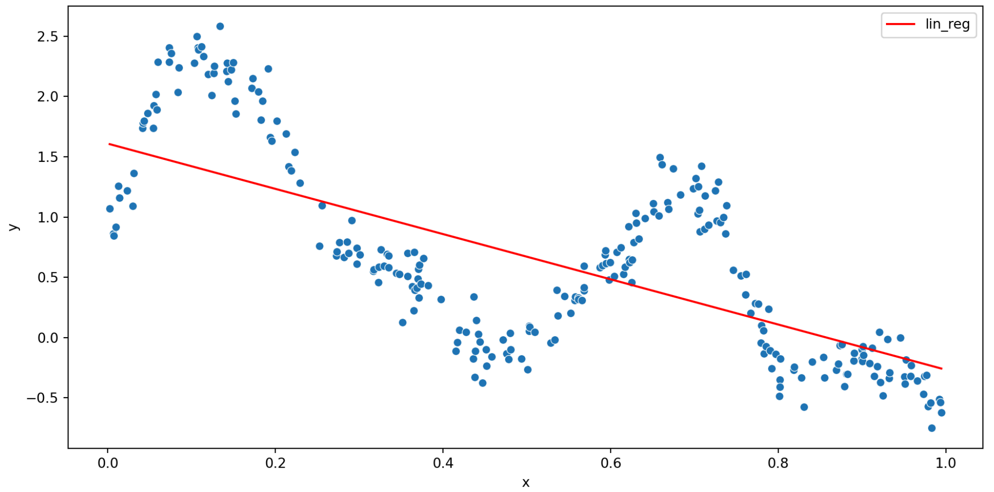
Model results

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

Training loss:



Predictions



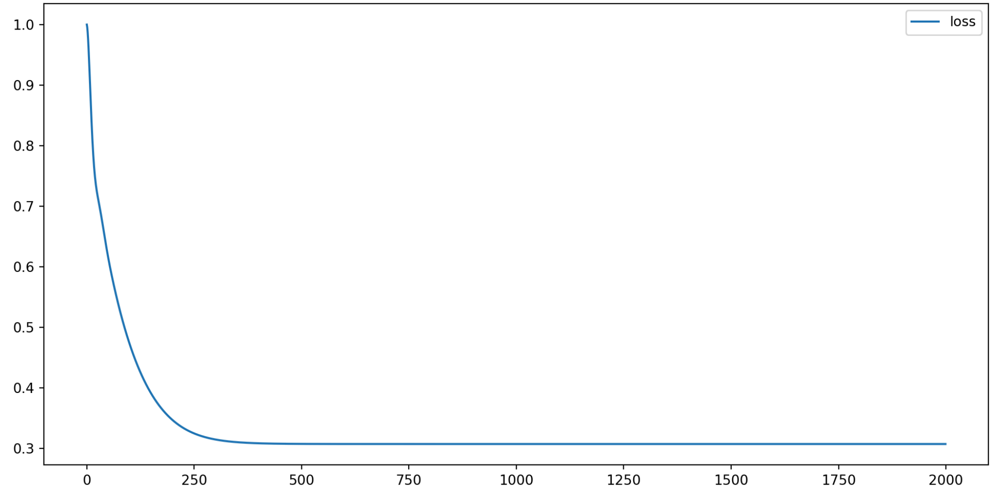
Double linear regression

```
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())

      return losses
```

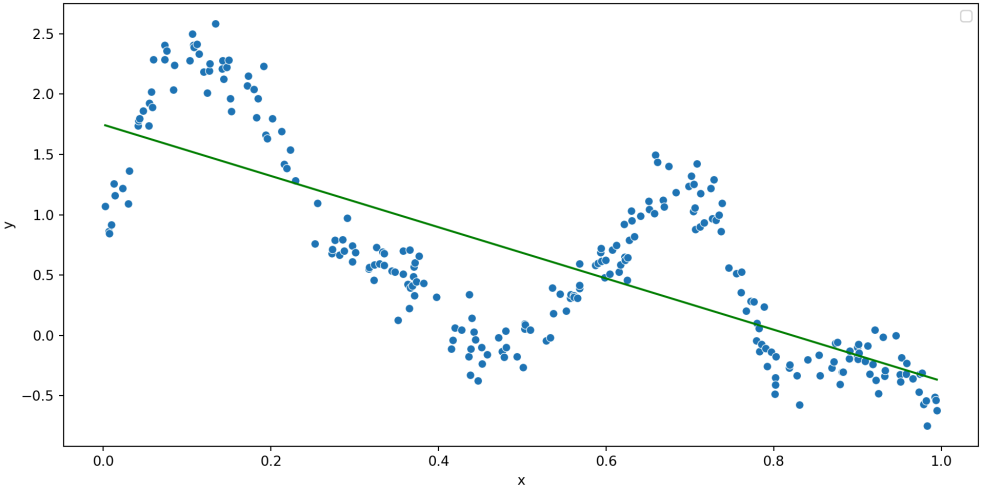
Model results

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

Training loss:



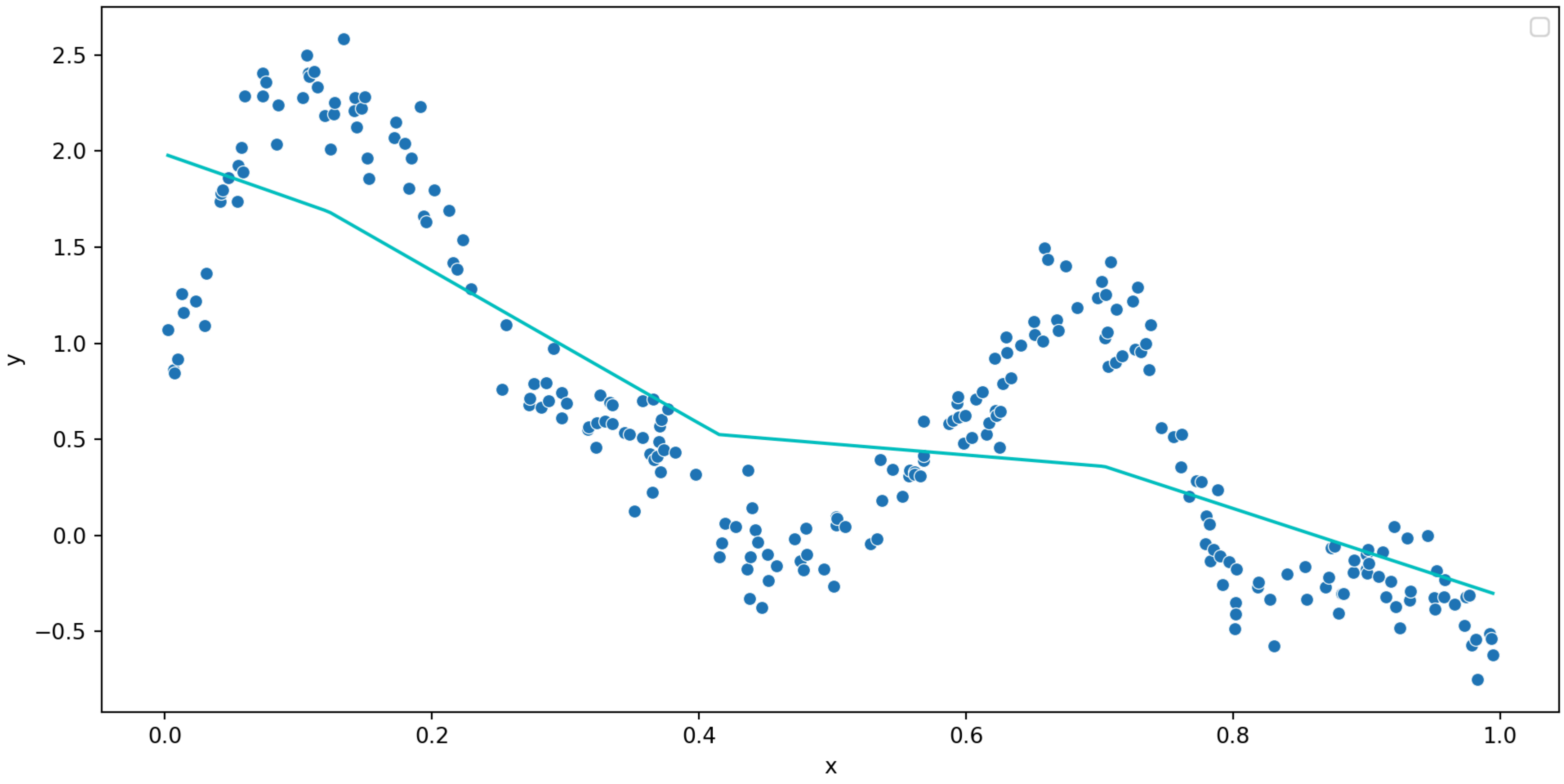
Predictions



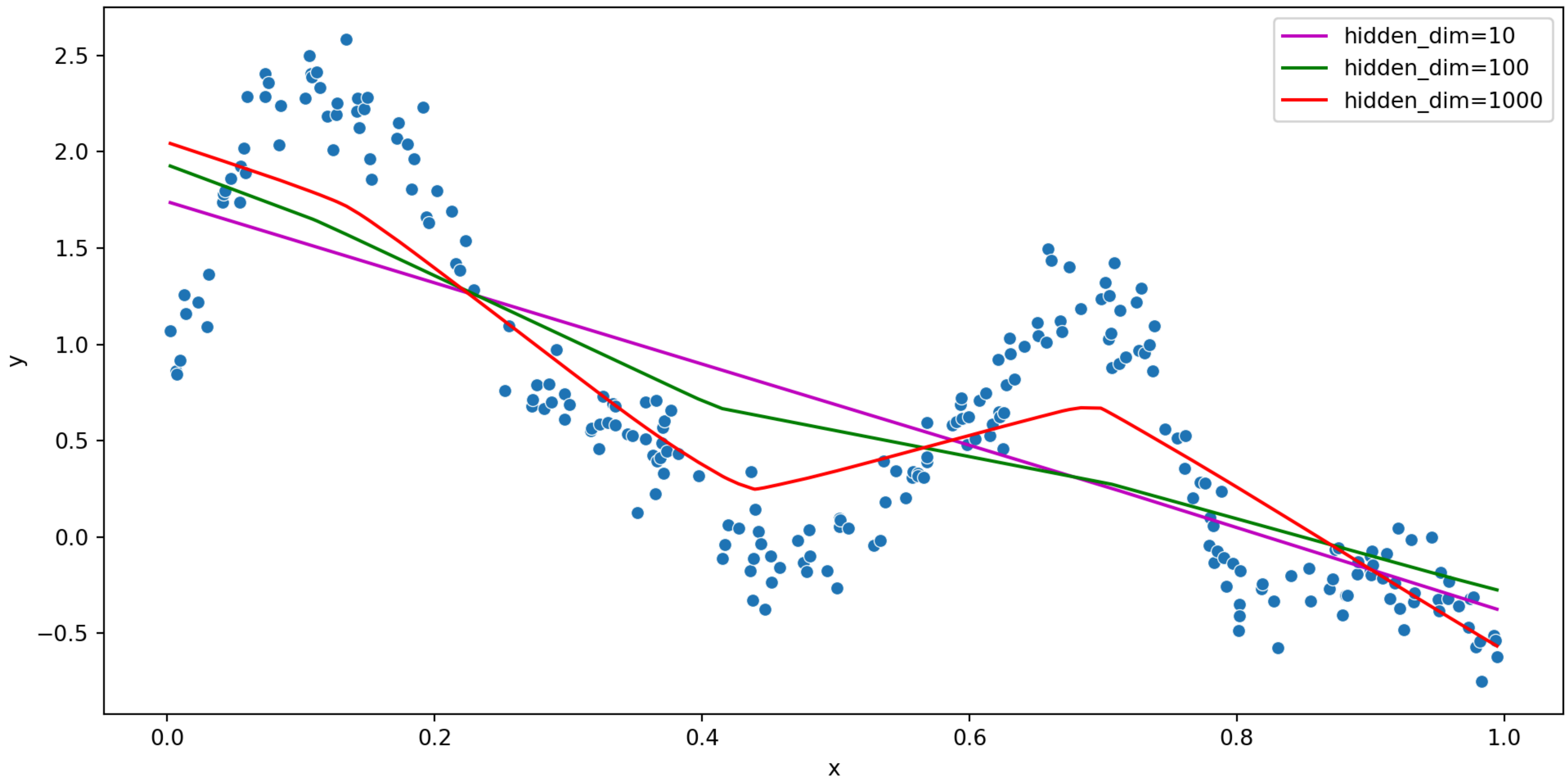
Non-linear regression w/ ReLU

```python
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



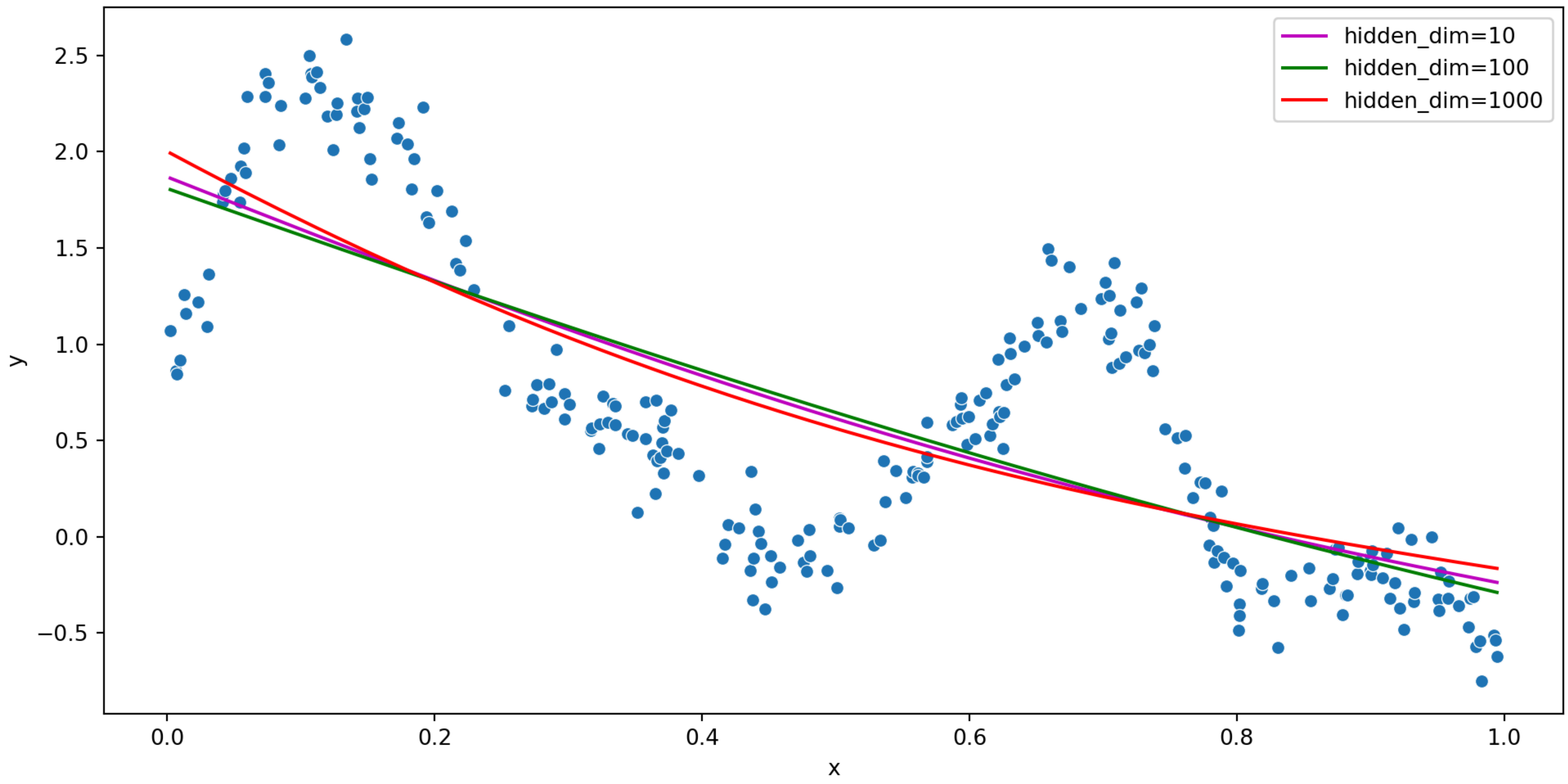
Hidden dimensions



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())

```

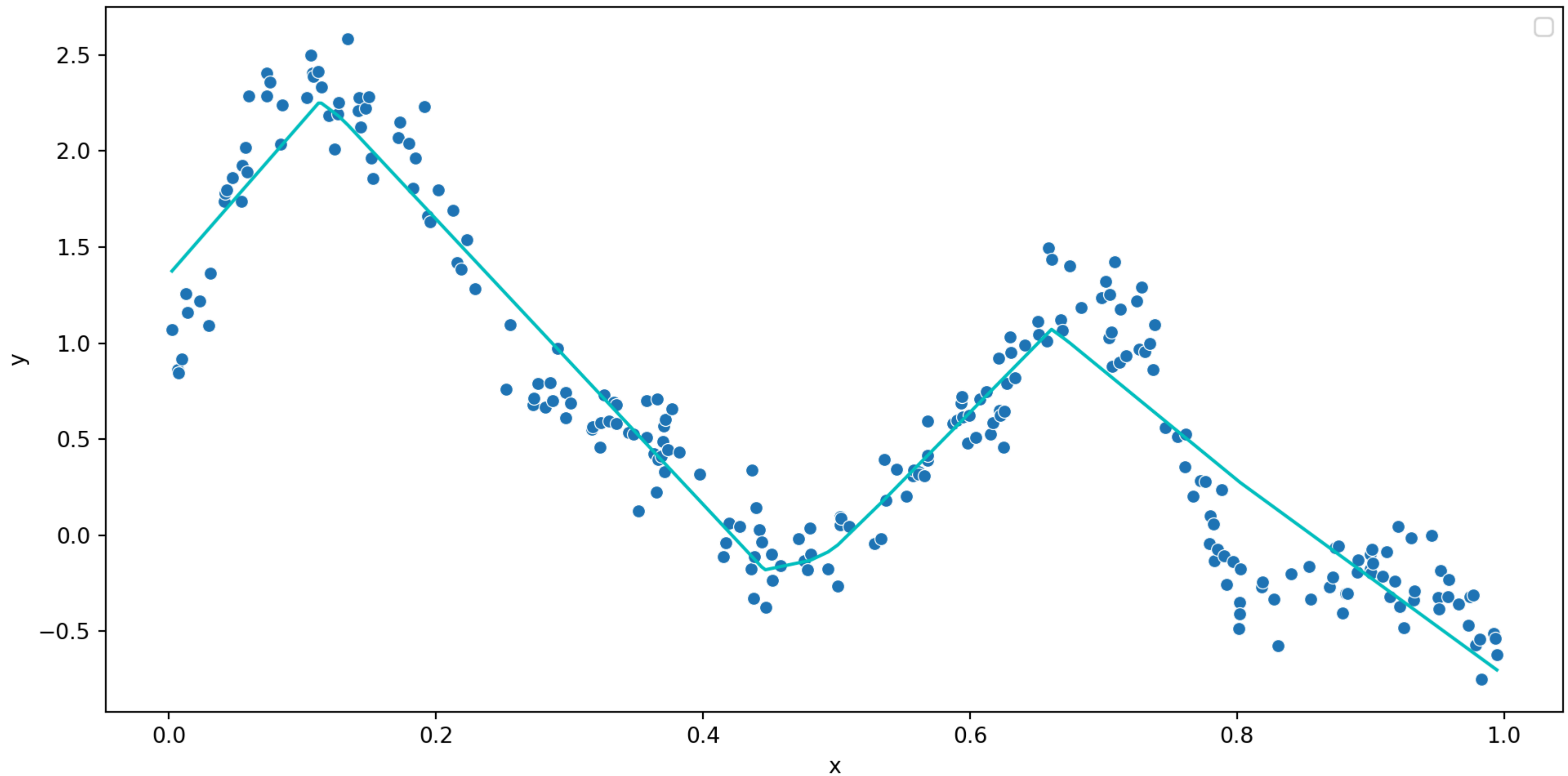
Tanh & hidden dimension



Three layers

```python
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()
          opt.zero_grad()
```

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)
      for i in range(n):
```

Model results