Dalle Molle Institute for Artificial Intelligence USI – SUPSI

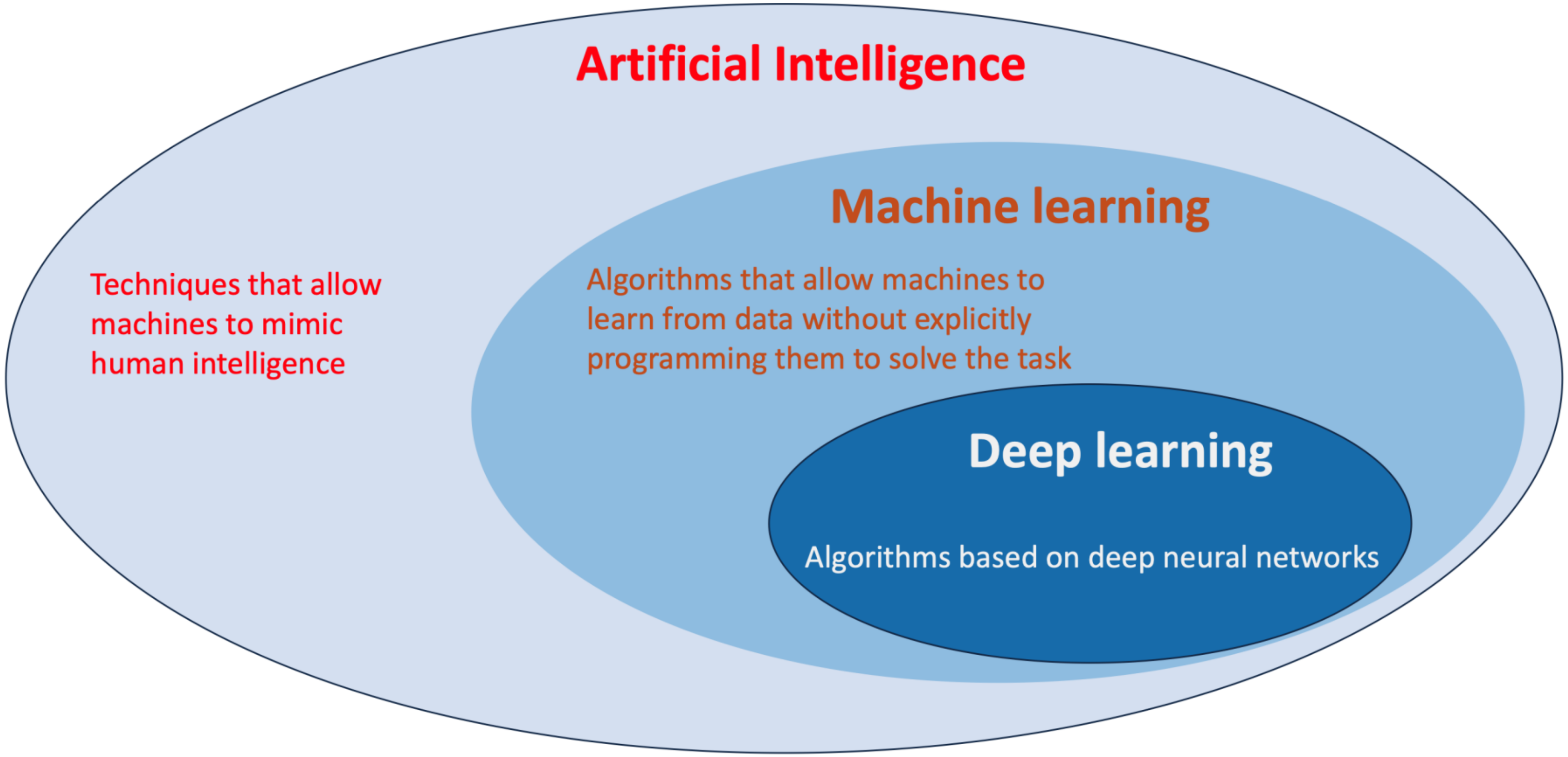
Feed-forward Neural Networks

Marco Forgione
SUPSI – Scuola Universitaria Professionale della Svizzera Italiana
IDSIA – Dalle Molle Institute for Artificial Intelligence

marco.forgione@supsi.ch

SUPSI

Artificial Intelligence, Machine Learning, Deep Learning

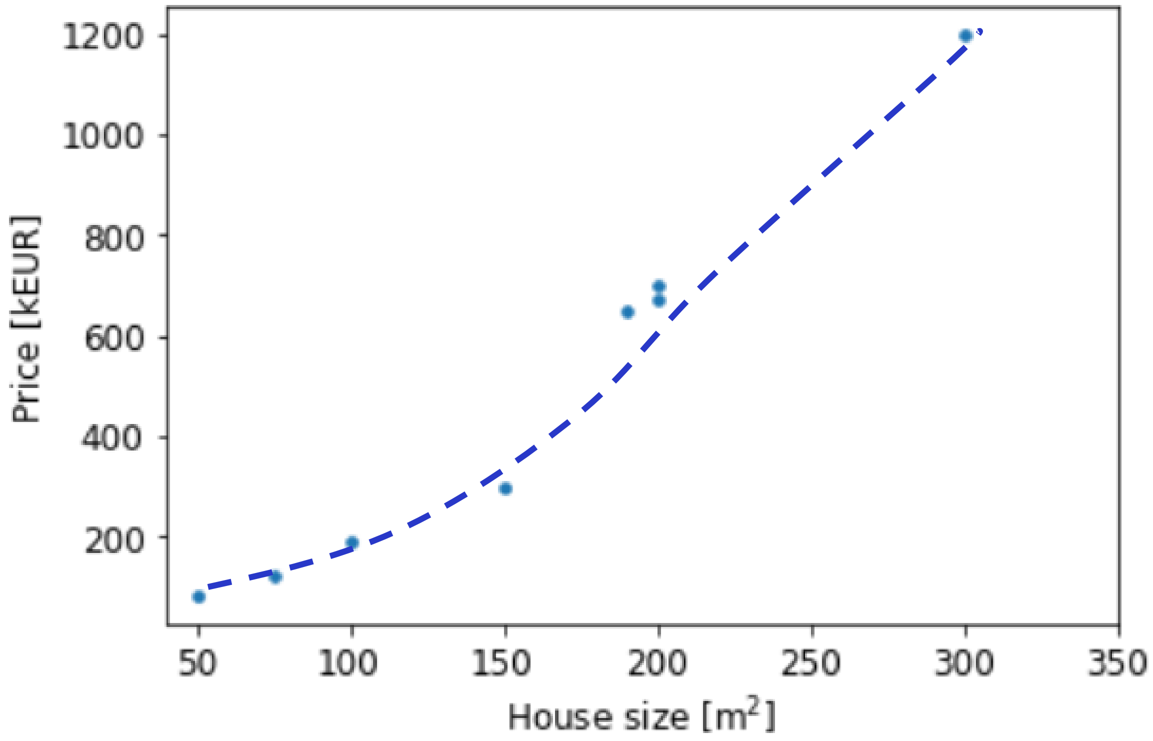


Machine learning: Regression

- $\{(x^i, y^i)\}\ \ i = 1, \ldots, N$ available dataset
- $\hat{y}^i = M(x^i; \theta)$ parametric model
- $L(\theta) = \frac{1}{N} \sum_{i=1}^{N} \left(y^i - \hat{y}^i(\theta)\right)^2$ Loss (MSE)

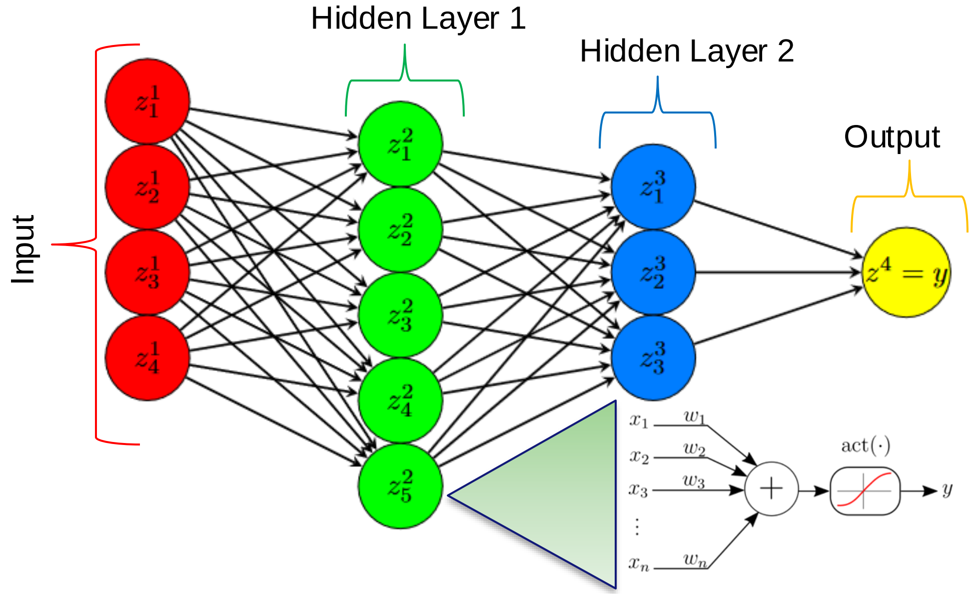
$$\boxed{\theta^* = arg \min_{\theta} L(\theta)}$$

Real estate application



Feed-Forward Neural Networks

It is just a particular model structure



$$z_1^2 = \sigma\left(\sum_{j=1}^{4} w_{1,j}^1 z_j^1 + b_1^1\right) = \sigma\left(W_1^1 z^1 + b_1^1\right)$$

$$z_2^3 = \sigma\left(\sum_{j=1}^{5} w_{2,j}^2 z_j^2 + b_2^2\right) = \sigma\left(W_2^2 z^2 + b_2^2\right)$$

$$y = z^4 = \sum_{j=1}^{3} w_{1,j}^3 z_j^3 + b^3 = W_1^3 z^3 + b^3$$

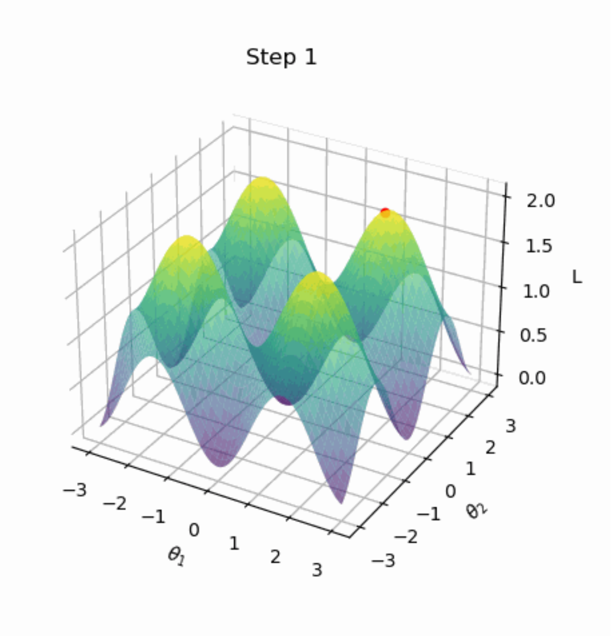
$$y = z^4 = f(z^3(z^2(z^1)); W, b)$$

For regression, the output neuron typically does not have a nonlinear activation function

Vanilla Gradient Descent

1. **Initialize Parameters**: $\theta^{(0)}$
2. **for** $k = 0, 1, \ldots$: **until** maximum number of iterations or convergence **do**:
   (a) **Compute Gradient**: $\nabla_\theta \mathcal{L}(\theta^{(k)})$
   (b) **Update Parameters**:

$$\theta^{(k+1)} = \theta^{(k)} - \gamma \cdot \nabla_\theta \mathcal{L}(\theta^{(k)})$$




```
optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.9)
```

Nowadays, the adam optimizer (optim.Adam) is a more common variant of vanilla gradient descent.

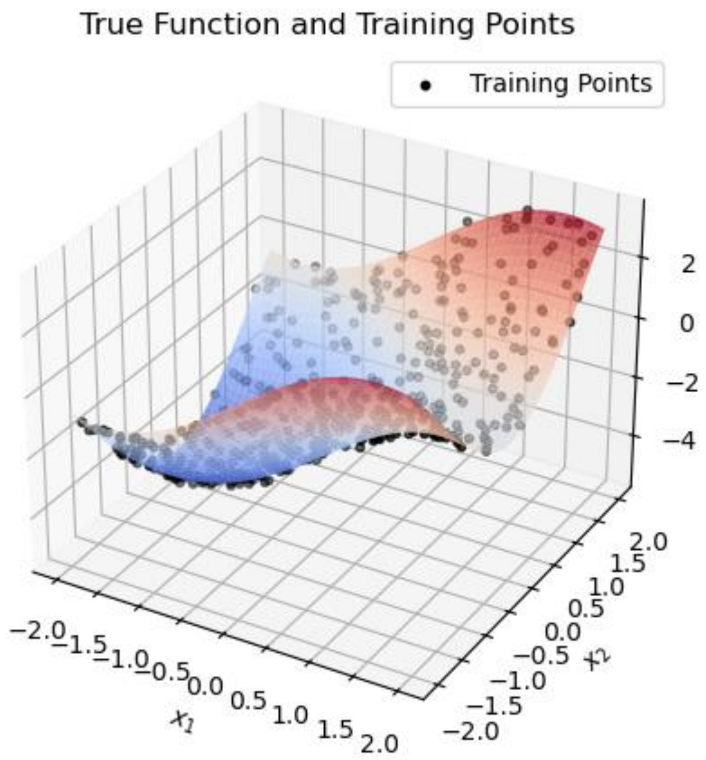
Exercise: synthetic toy dataset (toy_example.ipynb)

Consider the 2D function $f : \mathbb{R}^2 \to \mathbb{R}$

$$f(x) = 2\sin(x_1) - 3\cos(x_2)$$

$$x \in [-2, 2]^2 \subset \mathbb{R}^2$$

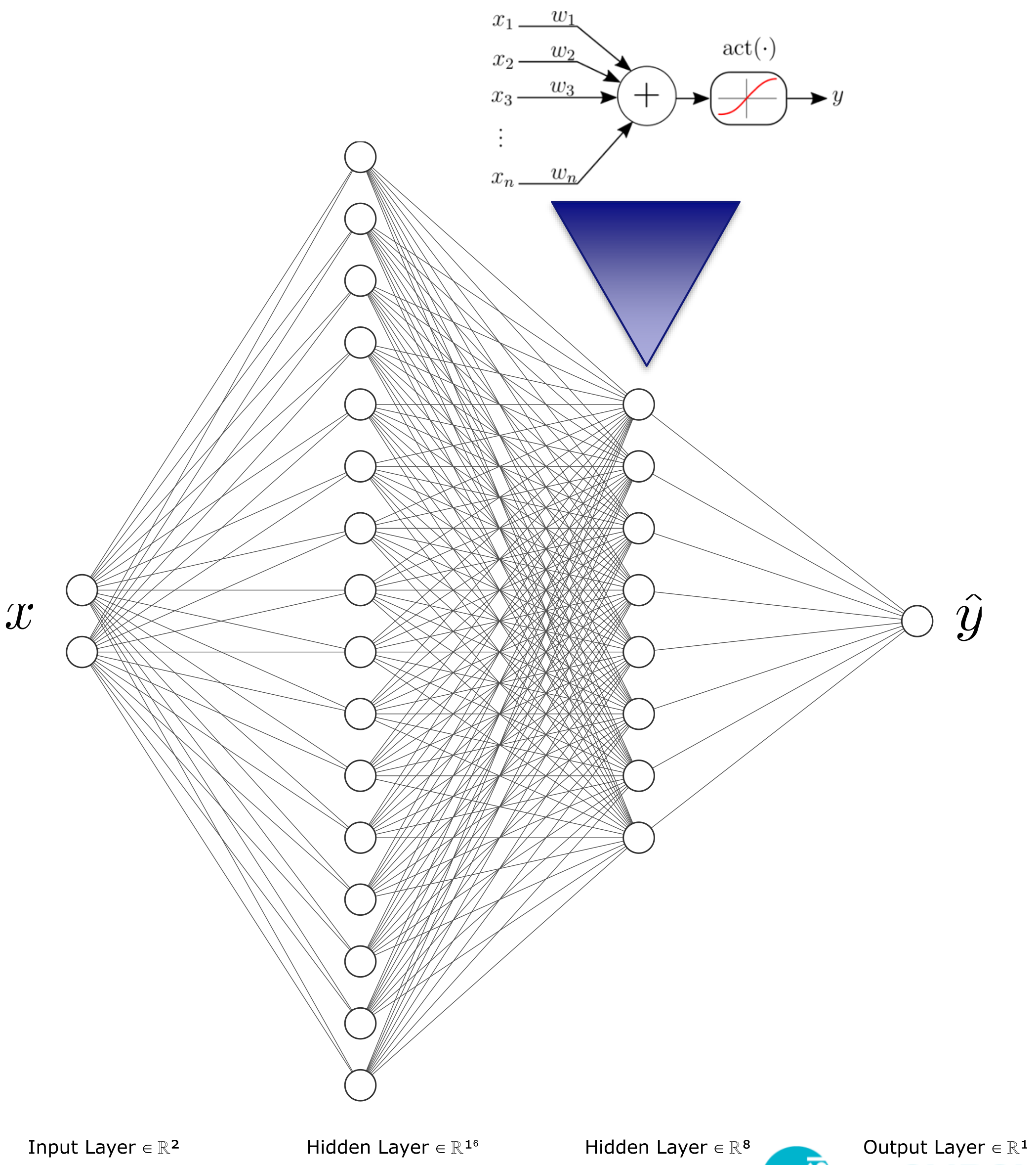
- Training and test datasets: 500 points uniformly sampled in the domain.
- Additive noise with standard deviation 0.1



Feed-forward neural network model

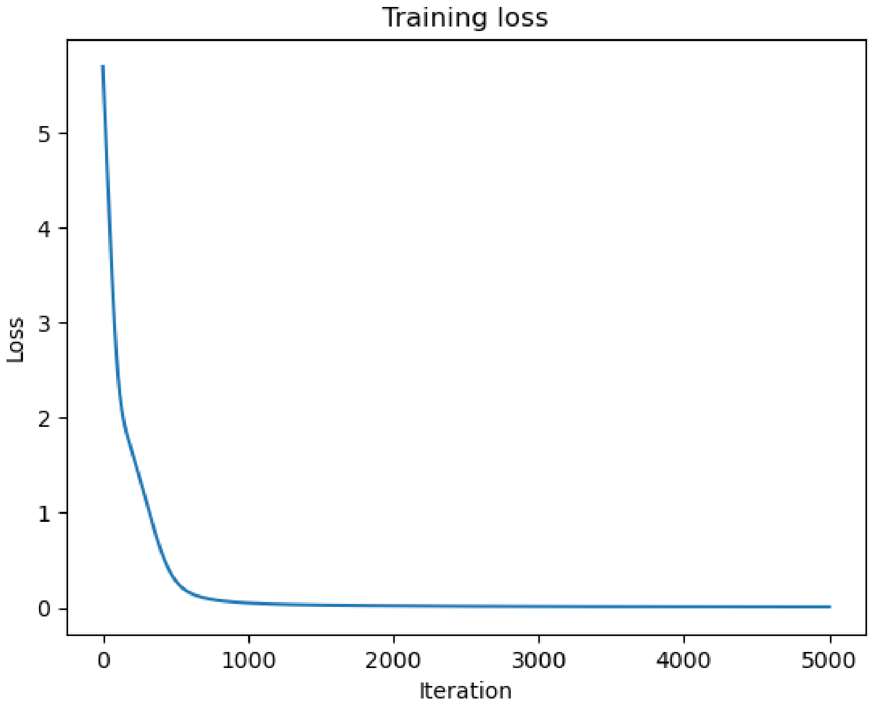
- 2 inputs, 1 output - this is the structure of $f(x)$
- 2 hidden layers with [16, 8] neurons
- tanh non-linearities

```python
class FeedForwardNN(nn.Module):
    def __init__(self):
        super(FeedForwardNN, self).__init__()
        self.fc1 = nn.Linear(2, 16)
        self.act1 = nn.Tanh()
        self.fc2 = nn.Linear(16, 8)
        self.act2 = nn.Tanh()
        self.fc3 = nn.Linear(8, 1)

    def forward(self, x):
        x = self.act1(self.fc1(x)) # 16
        x = self.act2(self.fc2(x)) # 8
        x = self.fc3(x) # 1
        return x

model = FeedForwardNN()
```



Model training

```python
criterion = nn.MSELoss()
optimizer = optim.Adam(model.parameters(), lr=1e-3)

# Train the model
iters = 5000
LOSS = []
for iter in range(iters):
    optimizer.zero_grad()
    y_pred = model(X_train)
    loss = criterion(y_pred, y_train)
    loss.backward()
    optimizer.step()

    LOSS.append(loss.item())
    if iter % 100 == 0:
        print(f"Epoch {iter}: Loss = {loss.item():.4f}")
```



Model evaluation

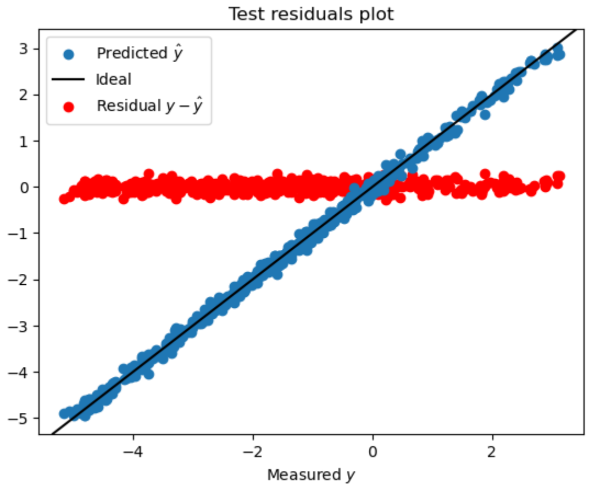
It is common to inspect, on the test dataset:

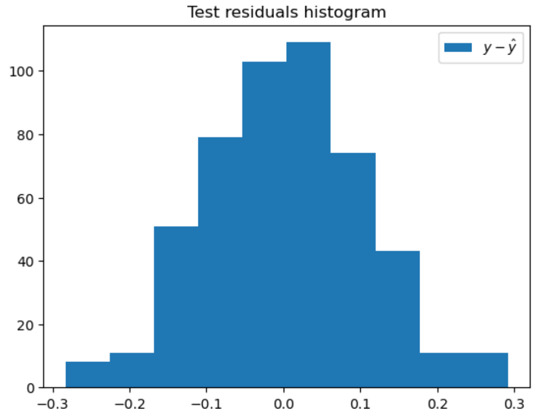
- Predictions and residuals vs measured y (top left)
- Histogram of residuals (top right)

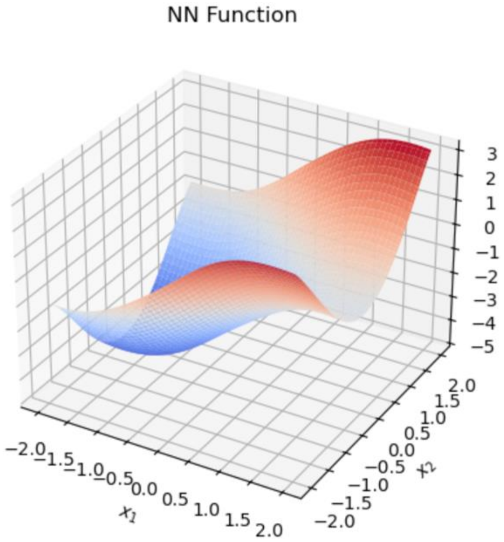
In this toy example, we can do more:

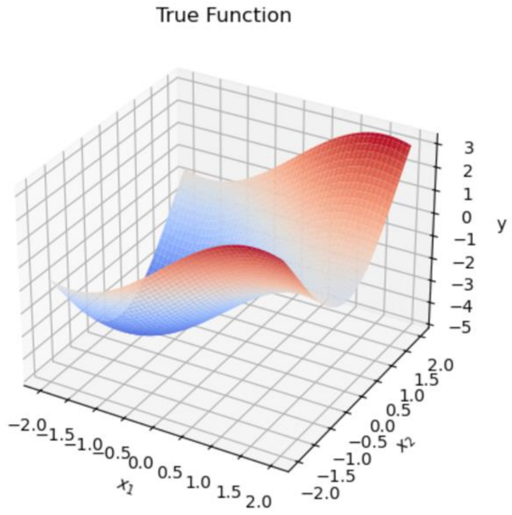
- The input is only 2D. We can visualize the learned function over a grid (bottom left)
- The true function is known. We can also visualize it in 2D (bottom right)

Everything seems to work well!







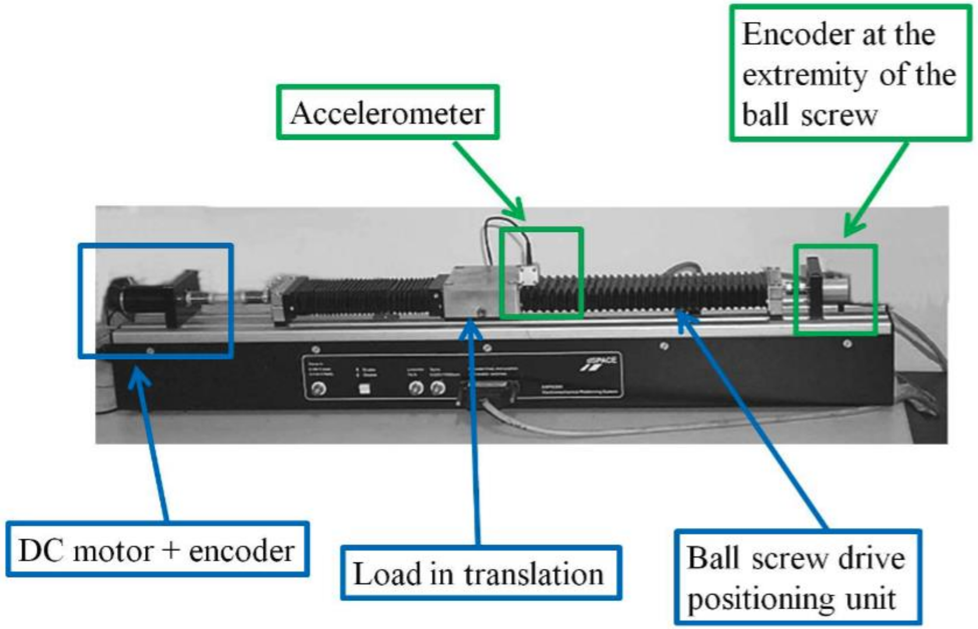


Inverse Dynamical Modeling on the EMPS benchmark (emps_exercise.ipynb)

Real dataset from a (simple) mechanical system: the Electro-Mechanical Positioning System

- Prismatic joint, 1-DoF mechanical system

$$M\ddot{q}(t) = \tau(t) - \tau_f(t) - b$$

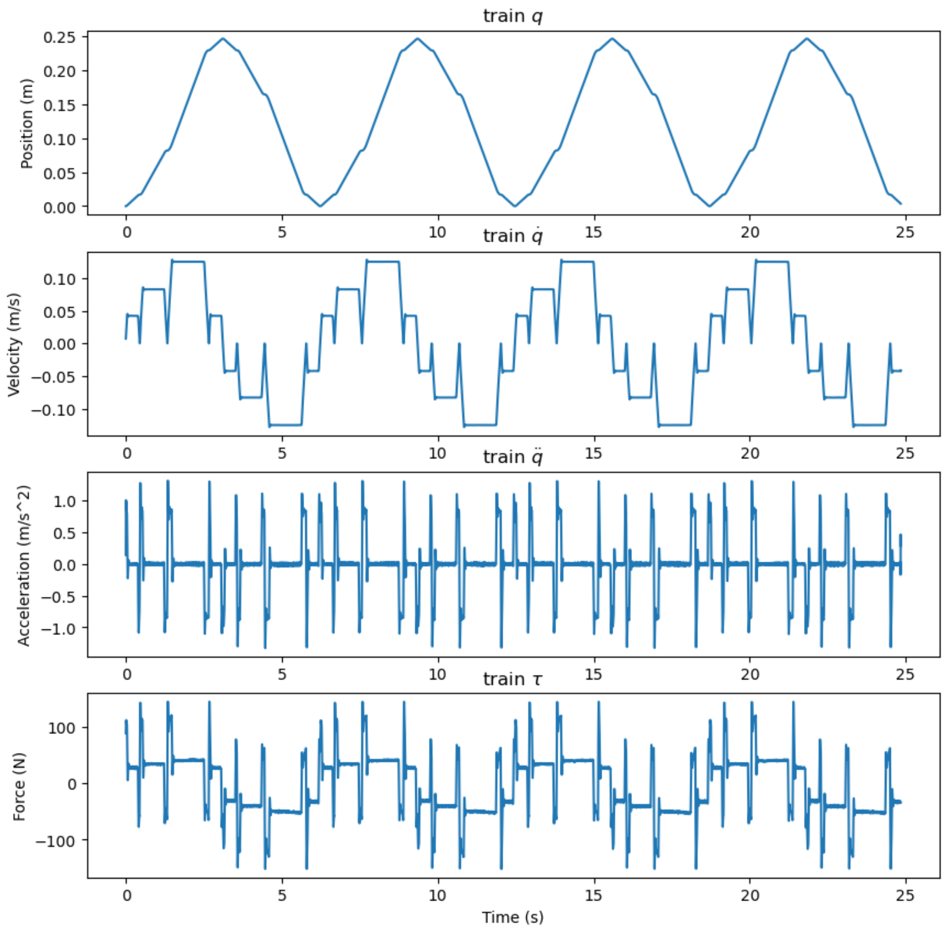


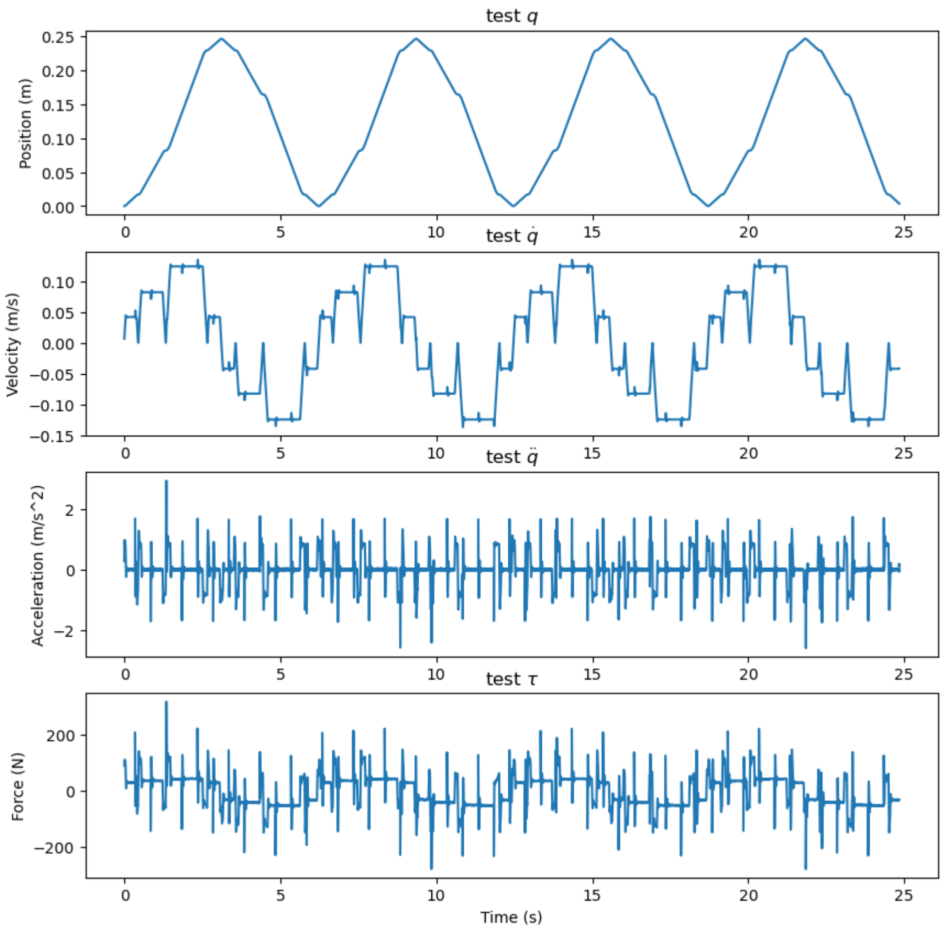
- $q(t)$: measured position (m)
- $\tau(t)$: known applied force (N)
- $\tau_f(t)$: unknown friction (N)
- M: unknown mass (kg)
- b: force measurement bias (N)

We want an inverse dynamical model (IDM): $q(t), \dot{q}(t), \ddot{q}(t) \rightarrow \tau(t)$

Dalle Molle Institute for Artificial Intelligence USI – SUPSI

Training and test datasets





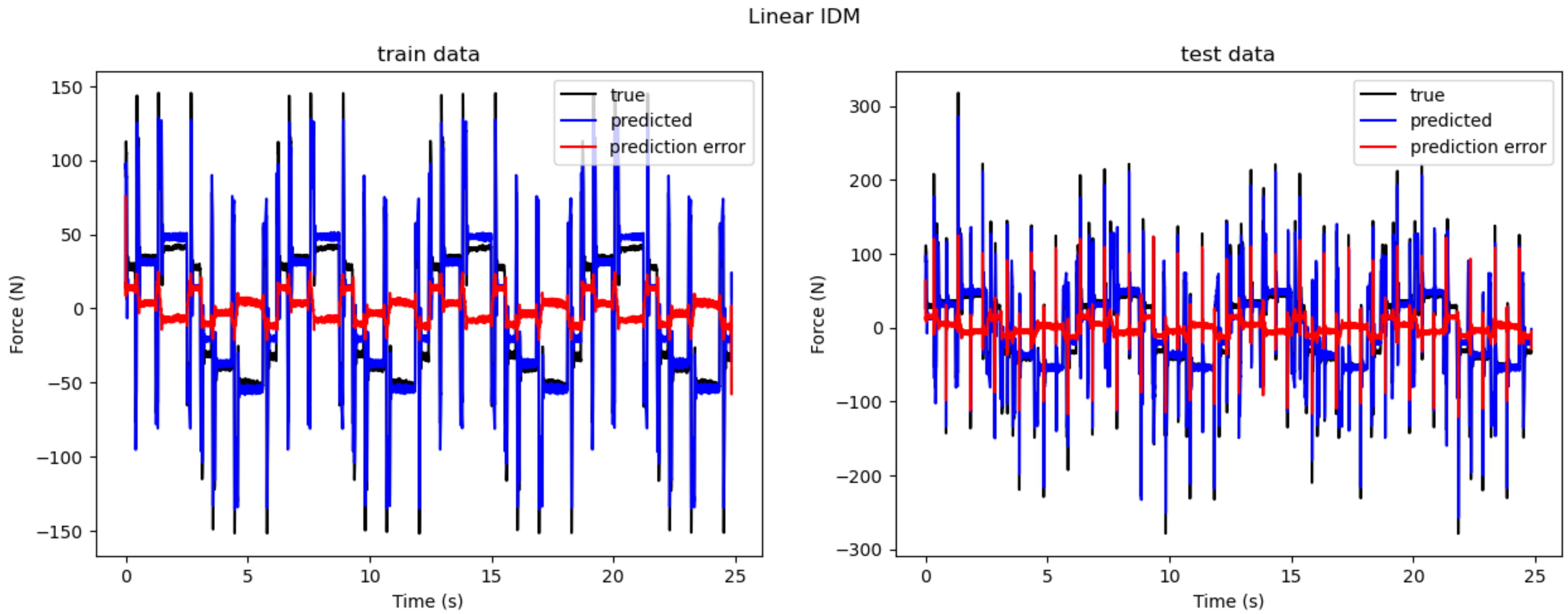
SUPSI

Linear model

We assume a linear friction model: $\tau_f = -F_v \dot{q}(t)$

Then, the IDM is: $\tau(t) = M\ddot{q}(t) + F_v \dot{q}(t) + b$. We can fit the IDM with a **linear regression**:

$$\tau(t) = \phi(t)\theta, \qquad \phi(t) = [\ddot{q}(t)\;\dot{q}(t)\;1], \qquad \theta = [M\;F_v\;b]^\top$$

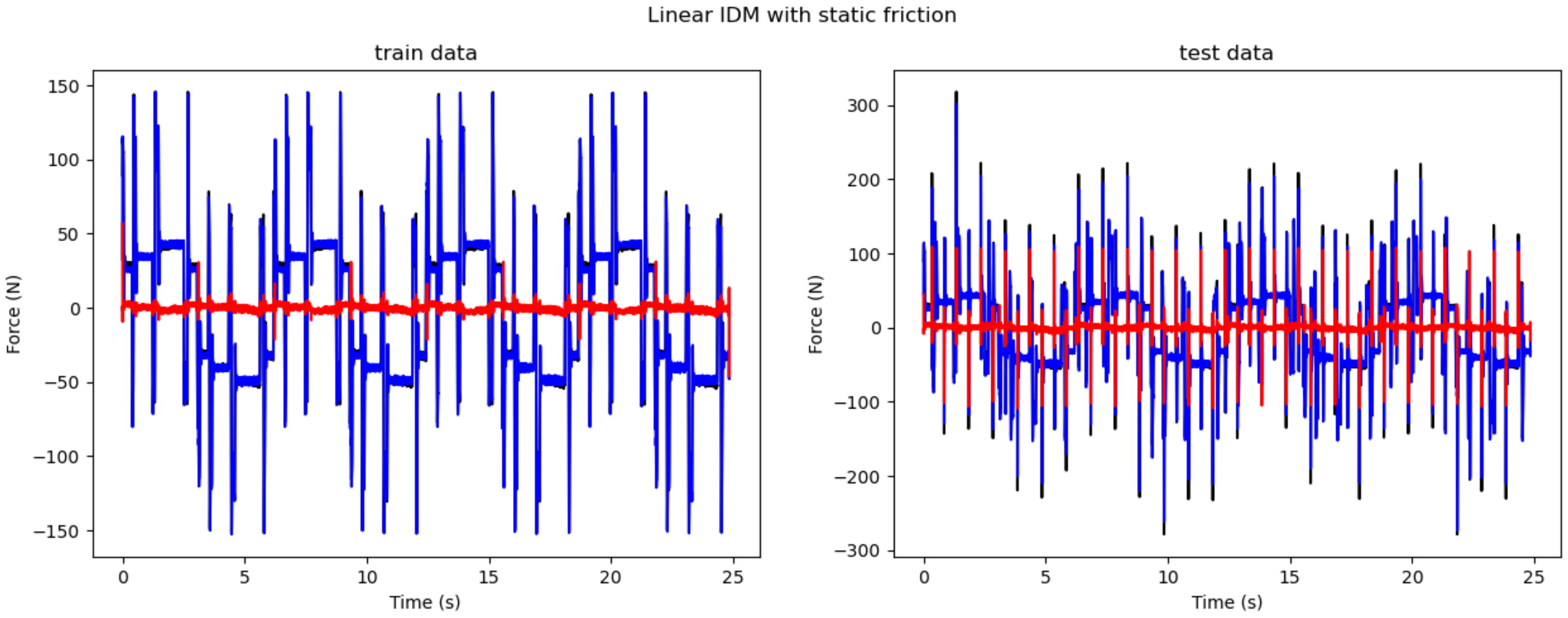


Dalle Molle Institute for Artificial Intelligence USI – SUPSI

Linear model with static friction

We use a more sophisticated friction model: $\tau_f(t) = -F_v \dot{q}(t) - F_c \mathrm{sign}(\dot{q}(t))$

$$\tau(t) = \phi(t)\theta, \qquad \phi(t) = [\ddot{q}(t)\,\dot{q}(t)\,\mathrm{sign}(q(t))\,1], \qquad \theta = [M\;F_v\;F_c\;b]^\top$$

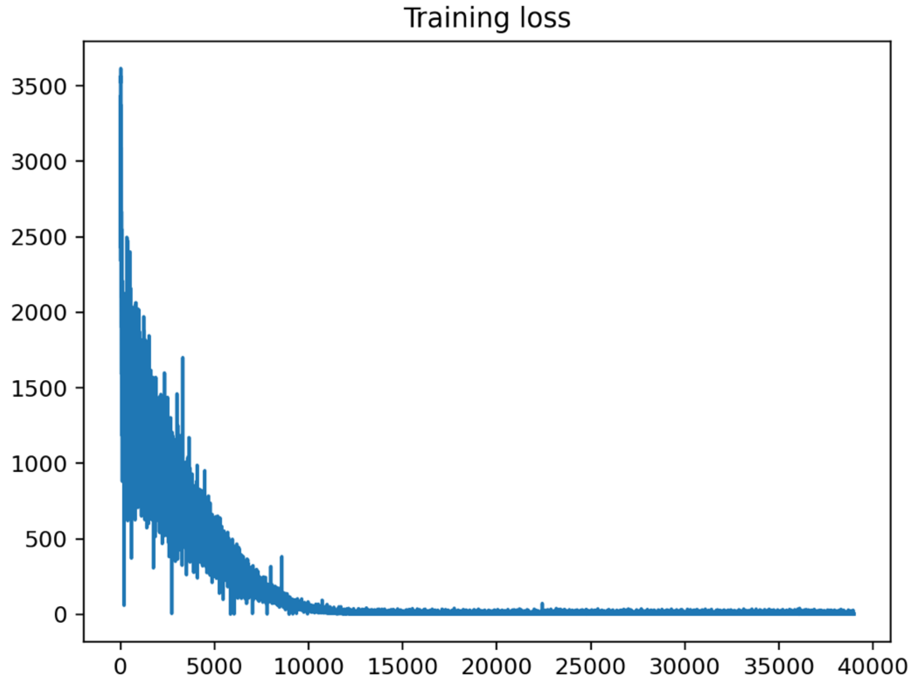


SUPSI

Feed-forward neural network

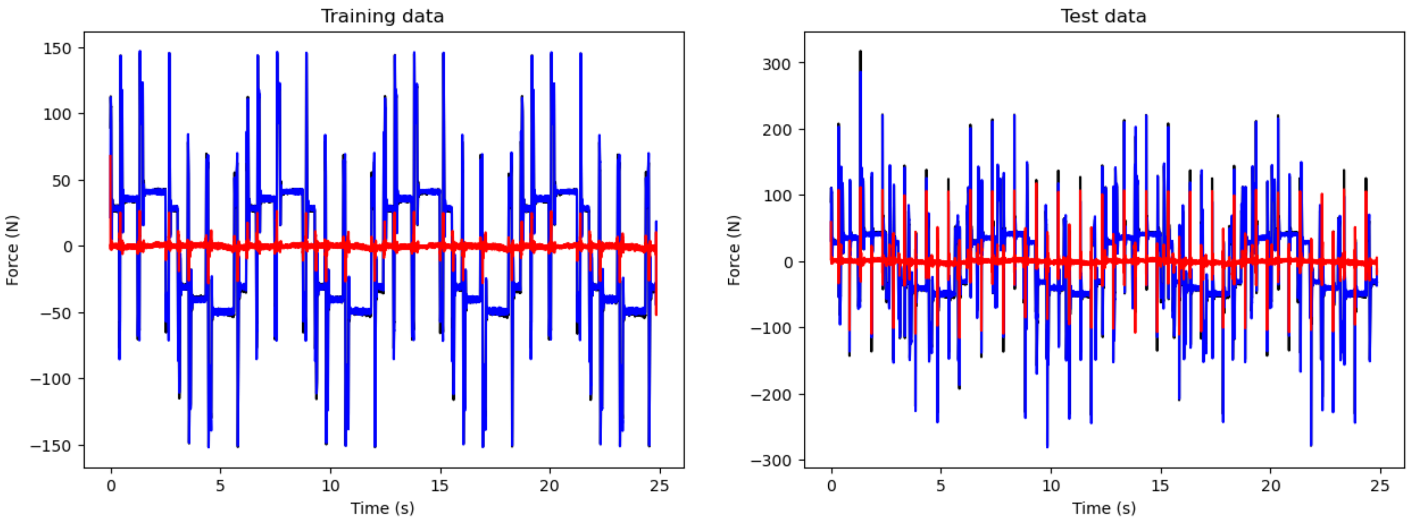
We ignore all physics and fit a feed-forward neural network instead: $\tau(t) = \mathrm{FF}(\ddot{q}, \dot{q})$

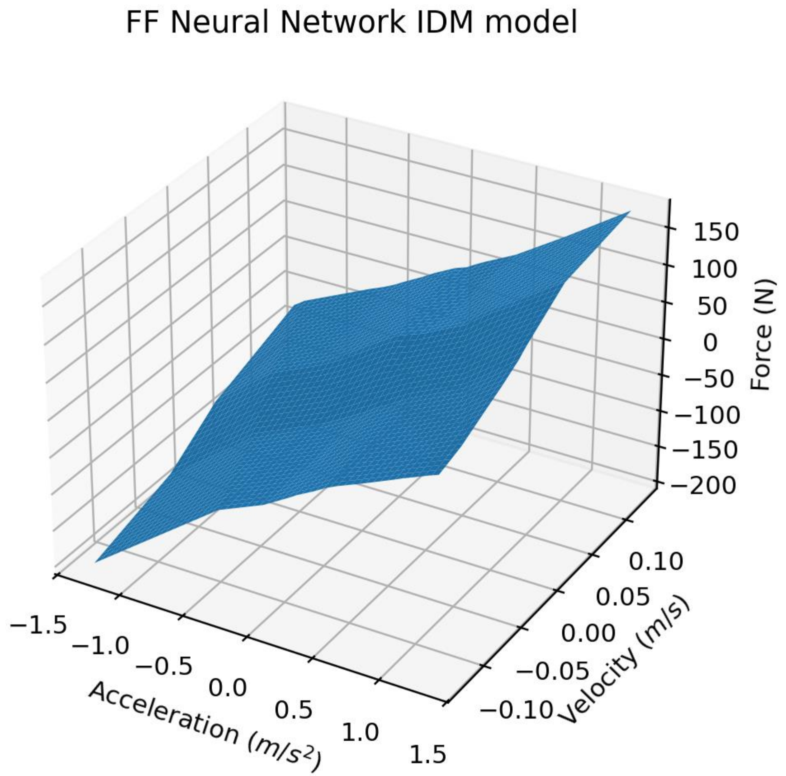
```
in_dim = train_X_torch.shape[1] # 2
out_dim = 1
batch_size = 128
hidden_size = 32
lr = 5e-4

friction_net = torch.nn.Sequential(
    torch.nn.Linear(in_dim, hidden_size),
    torch.nn.ReLU(),
    torch.nn.Linear(hidden_size, hidden_size),
    torch.nn.ReLU(),
    torch.nn.Linear(hidden_size, 1)
)
```



Train it yourself!

Feed-forward neural network



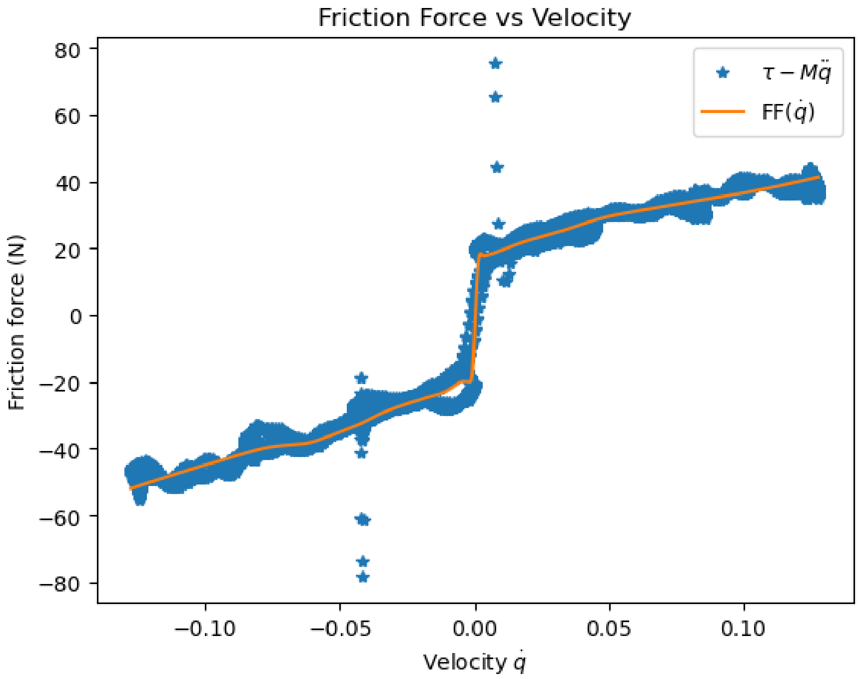


Train it yourself!

Physics-inspired neural network

We trust Newton's law, but nothing else: $\tau(t) = M\ddot{q} + \mathrm{FF}(\dot{q})$

```
class CustomIDM(nn.Module):
    def __init__(self, n_q=1, hidden_size=32):
        self.nq = n_q
        super(CustomIDM, self).__init__()
        self.inertia_net = nn.Linear(n_q, self.nq, bias=False)
        self.friction_net = nn.Sequential(
            nn.Linear(n_q, hidden_size),
            nn.GELU(),
            nn.Linear(hidden_size, hidden_size),
            nn.GELU(),
            nn.Linear(hidden_size, 1)
        )

    def forward(self, x):
        inertia = self.inertia_net(x[:, :self.nq])  # Linear in \ddot q
        friction = self.friction_net(x[:, self.nq:])  # Non-linear in \dot q
        return inertia + friction
```



Train it yourself!