Deep Learning with JAX

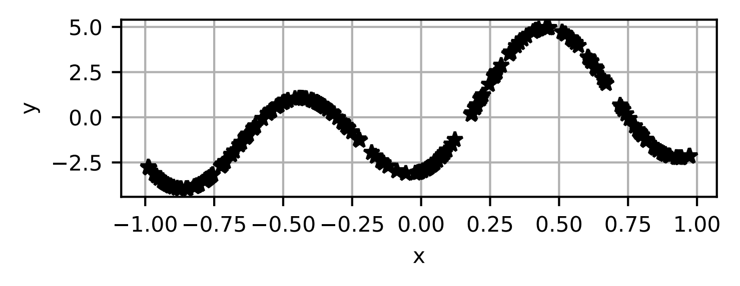
Feedforward neural networks

Marco Forgione

IDSIA USI-SUPSI, Lugano, Switzerland

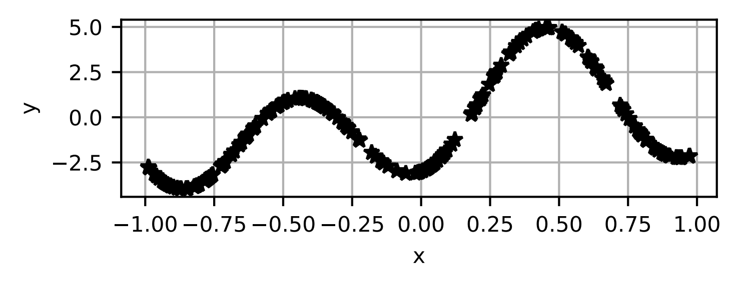
A non-linear dataset

Linear regression is not that exciting. Let's fit a neural net to a non-linear dataset!



A non-linear dataset

Linear regression is not that exciting. Let's fit a neural net to a non-linear dataset!




```
x_train.shape, y_train.shape
```

```
((200, 1), (200, 1))
```

Feed-forward neural network

- A one-hidden-layer feedforward neural network:

$$y = W_2 \tanh(W_1 x + b_1) + b_2$$

with tunable weights W_1, b_1, W_2, b_2.

Feed-forward neural network

- A one-hidden-layer feedforward neural network:

$$y = W_2 \tanh(W_1 x + b_1) + b_2$$

  with tunable weights W_1, b_1, W_2, b_2.

- If we prefer a single parameter vector:

$$p = \mathrm{vec}(W_1, b_1, W_2, b_2), \quad W_1 \in \mathbb{R}^{n_h \times n_x}, b_1 \in \mathbb{R}^{n_h}, W_2 \in \mathbb{R}^{n_y \times n_h}, b_2 \in \mathbb{R}^{n_y}.$$

Feed-forward neural network

- A one-hidden-layer feedforward neural network:

$$y = W_2 \tanh(W_1 x + b_1) + b_2$$

  with tunable weights W_1, b_1, W_2, b_2.

- If we prefer a single parameter vector:

$$p = \mathrm{vec}(W_1, b_1, W_2, b_2), \;\; W_1 \in \mathbb{R}^{n_h \times n_x}, b_1 \in \mathbb{R}^{n_h}, W_2 \in \mathbb{R}^{n_y \times n_h}, b_2 \in \mathbb{R}^{n_y}.$$

- The feedforward net is just a non-linear function:

$$y = f(p, x).$$

Feed-forward neural network

- A one-hidden-layer feedforward neural network:

$$y = W_2 \tanh(W_1 x + b_1) + b_2$$

  with tunable weights W_1, b_1, W_2, b_2.

- If we prefer a single parameter vector:

$$p = \mathrm{vec}(W_1, b_1, W_2, b_2), \;\; W_1 \in \mathbb{R}^{n_h \times n_x}, b_1 \in \mathbb{R}^{n_h}, W_2 \in \mathbb{R}^{n_y \times n_h}, b_2 \in \mathbb{R}^{n_y}.$$

- The feedforward net is just a non-linear function:

$$y = f(p, x).$$

In our SISO problem $n_x = 1, n_y = 1$. We can choose n_h (hyper-parameter).

Feed-forward neural network

In code, it is convenient to organize parameters in a *dictionary*

```
def nn(p, x):
    h = jnp.tanh(p["W1"] @ x + p["b1"])
    y = p["W2"] @ h + p["b2"]
    return y
```

$$y = W_2 \overbrace{\tanh(W_1 x + b_1)}^{=h} + b_2$$

Feed-forward neural network

In code, it is convenient to organize parameters in a *dictionary*

```
def nn(p, x):
    h = jnp.tanh(p["W1"] @ x + p["b1"])
    y = p["W2"] @ h + p["b2"]
    return y
```

$$y = W_2 \overbrace{\tanh(W_1 x + b_1)}^{=h} + b_2$$

The parameter dictionary may be initialized like this:

```
nx = 1; ny = 1; nh = 16
params = {
  "W1": jr.normal(key_W1, shape=(nh, nx)),
  "b1": jr.normal(key_b1, shape=(nh,)),
  "W2": jr.normal(key_W2, shape=(ny, nh)),
  "b2": jr.normal(key_b2, shape=(ny,)),
}
```

Feed-forward neural network

In code, it is convenient to organize parameters in a *dictionary*

```
def nn(p, x):
    h = jnp.tanh(p["W1"] @ x + p["b1"])
    y = p["W2"] @ h + p["b2"]
    return y
```

$$y = W_2 \overbrace{\tanh(W_1 x + b_1)}^{=h} + b_2$$

The parameter dictionary may be initialized like this:

```
nx = 1; ny = 1; nh = 16
params = {
  "W1": jr.normal(key_W1, shape=(nh, nx)),
  "b1": jr.normal(key_b1, shape=(nh,)),
  "W2": jr.normal(key_W2, shape=(ny, nh)),
  "b2": jr.normal(key_b2, shape=(ny,)),
}
```

Better initializations exist (Kaiming, Glorot,...). Key for big nets, omitted for simplicity.

Feed-forward neural network implementation

- Let us apply the neural network to a data point:

```
nn(params, x_train[0])
```

```
Array([4.4949026], dtype=float32)
```

Feed-forward neural network implementation

- Let us apply the neural network to a data point:

```
nn(params, x_train[0])
```

```
Array([4.4949026], dtype=float32)
```

- Our `nn` function only handles a single data point by construction.

```
nn(params, x_train) # x_train: (N, n_x). Does NOT work!
```

Feed-forward neural network implementation

- Let us apply the neural network to a data point:

```
nn(params, x_train[0])
```

```
Array([4.4949026], dtype=float32)
```

- Our `nn` function only handles a single data point by construction.

```
nn(params, x_train) # x_train: (N, n_x). Does NOT work!
```

- To process a *batch*, we must *vectorize* the `nn` function wrt its second argument.

```
# do nothing for first arg, add batch axis 0 for 2nd arg
batched_nn = jax.vmap(nn, in_axes=(None, 0))
```

Feed-forward neural network implementation

- Let us apply the neural network to a data point:

```
nn(params, x_train[0])
```

```
Array([4.4949026], dtype=float32)
```

- Our `nn` function only handles a single data point by construction.

```
nn(params, x_train) # x_train: (N, n_x). Does NOT work!
```

- To process a *batch*, we must *vectorize* the `nn` function wrt its second argument.

```
# do nothing for first arg, add batch axis 0 for 2nd arg
batched_nn = jax.vmap(nn, in_axes=(None, 0))
```

```
y_hat = batched_nn(params, x_train);
y_hat.shape
```

```
(200, 1)
```

Setting up the loss

- Define the loss as a function

```
def loss_fn(p, y, x):
  y_hat = batched_nn(p, x)
  loss = jnp.mean((y - y_hat) ** 2)
  return loss
```

Setting up the loss

- Define the loss as a function

```
def loss_fn(p, y, x):
  y_hat = batched_nn(p, x)
  loss = jnp.mean((y - y_hat) ** 2)
  return loss
```

- Define a function that return both loss and its gradient

```
loss_grad_fn = jax.value_and_grad(loss_fn, 0)
loss_grad_fn = jax.jit(loss_grad_fn) # compile it!
```

Setting up the loss

- Define the loss as a function

```
def loss_fn(p, y, x):
  y_hat = batched_nn(p, x)
  loss = jnp.mean((y - y_hat) ** 2)
  return loss
```

- Define a function that return both loss and its gradient

```
loss_grad_fn = jax.value_and_grad(loss_fn, 0)
loss_grad_fn = jax.jit(loss_grad_fn) # compile it!
```

```
loss_grad_fn(params, y_train, x_train)
```

```
(Array(24.432558, dtype=float32),
 {'W1': Array([[-0.9182231 ],
         [-0.32852545],
         [-0.28115115],
```

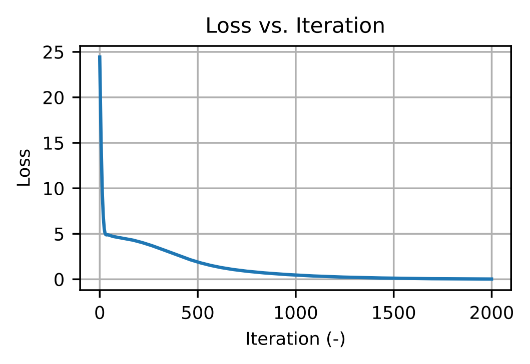
Fitting the neural net

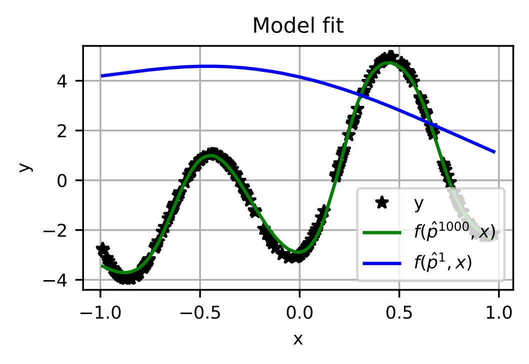
The *boilerplate* training code. Use optax instead of gradient descent from scratch...

```
# Define optimizer
optimizer = optax.adam(learning_rate=1e-2) # optax.{sgd, adam, ...}
opt_state = optimizer.init(params)
```

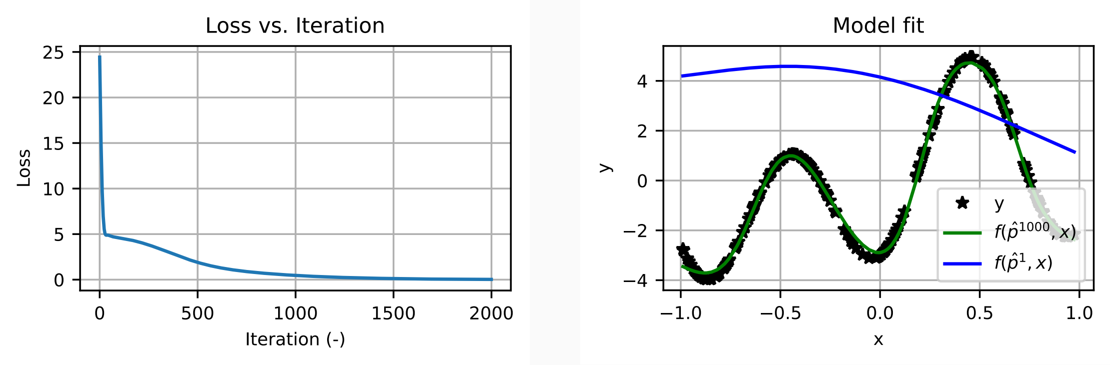
```
# Training loop
LOSS = []
for iter in range(2000):
    loss_val, grads = loss_grad_fn(params, y_train, x_train)

    updates, opt_state = optimizer.update(grads, opt_state)
    params = optax.apply_updates(params, updates)

    LOSS.append(loss_val)
```

Results


Loss vs. Iteration
Loss
25
20
15
10
5
0
0
500
1000
1500
2000
Iteration (-)

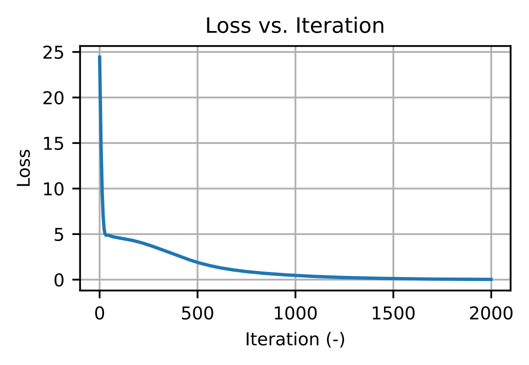

Model fit
y
4
2
0
−2
−4
−1.0
−0.5
0.0
0.5
1.0
x
y
$f(\hat{p}^{1000}, x)$
$f(\hat{p}^{1}, x)$

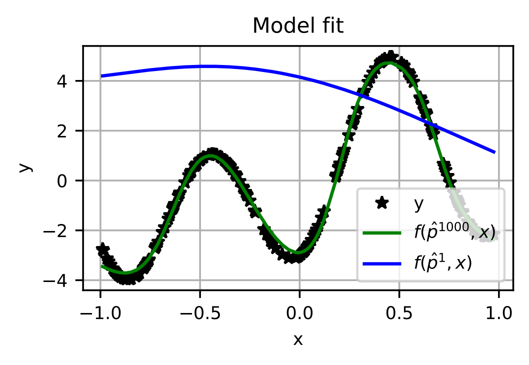
Results



- Good fit without knowing the model structure

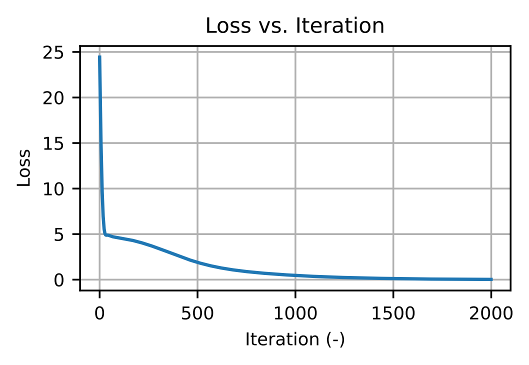
Results

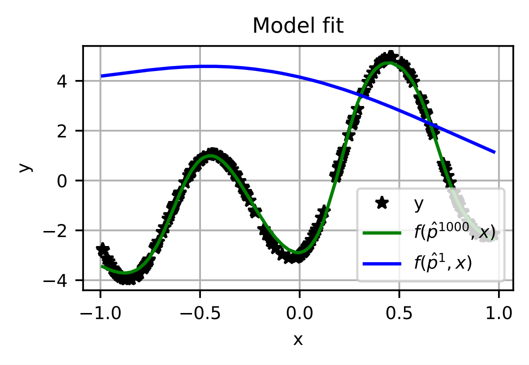




- Good fit without knowing the model structure
- Extension to MIMO is trivial - just change n_x, n_y

Results





- Good fit without knowing the model structure
- Extension to MIMO is trivial - just change n_x, n_y
- Extension to dynamical systems tomorrow

Exercises

- Try out different initialization and optimization settings
- Try out different hyper-parameters (e.g., hidden layers n_h, non-linearity function)
- Define and train a neural network with 2 hidden layers
- Replicate the 2D example in the introductory slides
- Implement mini-batching
    - At each optimization step, sample a *subset* of training instances