Deep Learning Lab - Lecture 3

Last modified: September 30, 2024

Instructor: E. Vercesi

TAs: A. Dei Rossi, G. Dominici, S. Huber

vercee@usi.ch - {alvise.dei.rossi, gabriele.dominici, stefano.huber} @usi.ch

September 30, 2024

Today plan/objective

- Linear regression with PyTorch
- Presentation of Assignment 1.
- You can start working on Assignment 1 and / or ask questions on Exercise 1 and / or ask questions on Exercise 2.

Toy task - Linear regression i

- Let $D, N \in \mathbb{N}_+$.
  - N data points
  - D dimension of the input
- Hence, our dataset is

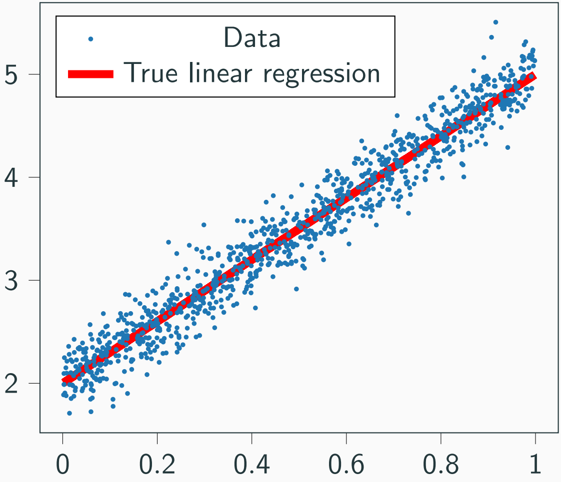
$$\mathcal{D} = \{(\boldsymbol{x}^1, y^1), \ldots, (\boldsymbol{x}^N, y^N) \mid \boldsymbol{x}^i \in \mathbb{R}^D,\ y^i \in \mathbb{R}\}$$

- Recall our goal: find $f_{\boldsymbol{\theta}}(\boldsymbol{x})$ that is a good approximation of the true function
- $\boldsymbol{\theta}$ should be *learned* from data

Toy task - Linear regression ii

- **Linear Regression**: $f_{\boldsymbol{\theta}}(\boldsymbol{x}) = \boldsymbol{w}^T\boldsymbol{x} + b,\ \boldsymbol{w} \in \mathbb{R}^D, b \in \mathbb{R}$
- **Model parameters:** $\boldsymbol{\theta} = [\boldsymbol{w}, b]$
- Set $\boxed{\mathrm{D} = 1}$ in all that follows, namely we want to learn a "line" in the common sense

Linear regression with $D = 1$



- Our linear model is $f_{w,b}(x) = wx + b$ where w, b should be learnt from data

Regression framework

This is a regression problem, we have to make some choices

- Loss function: Mean Squared Error

$$\textbf{MSE loss}: \mathcal{L}(w, b) := \frac{1}{N} \sum_{i=1}^{N} ||(wx^i + b) - y^i||_2^2$$

- Optimizer: Stochastic Gradient Descent
- Hyperparameters:
  (a) Learning rate α
  (b) Epochs E number of times to see datasets
  (c) Batch size B divide dataset into smaller

Note: you can compute the number of steps T as

$$T = \frac{EN}{B}$$

Today: no batches, no epochs, just steps!

The algorithm

In this case, it is simple to design the algorithm step-by-step: after choosing the hyperparameters

(1) Randomly initialize parameters w_0, b_0

(2) For each step $t \leq T$

   (i) Compute the loss $\mathcal{L}(w_t, b_t)$

   (ii) Compute the gradient

$$\left[\frac{\partial \mathcal{L}}{\partial w}(w_t, b_t), \frac{\partial \mathcal{L}}{\partial b}(w_t, b_t)\right]$$

   (iii) Update parameters:

$$w_{t+1} = w_t - \alpha \frac{\partial \mathcal{L}}{\partial w}(w_t, b_t)$$

   and

$$b_{t+1} = b_t - \alpha \frac{\partial \mathcal{L}}{\partial b}(w_t, b_t)$$

Toy example

- Toy example means that:
  - We *pretend to know* the real function
  - We generate data from the true function *adding noise*
  - We end up with noisy data points (reality has noise)
- To generate data:
  - Start with the true w^*, b^* (we choose them)
  - Randomly sample x^i
  - Add *Gaussian noise*, that is

$$y^i = w^* x^i + b^* + \varepsilon^i \qquad \varepsilon^i \sim \mathcal{N}(0, \sigma)$$

  - (x^i, y^i) are our noisy data point

A “create noisy data” function

```
def create_dataset(sample_size=10, sigma=0.1, w_star=1, b_star = 1,
                   x_range=(-1, 1), seed=0):
    # Set the random state in numpy
    torch.manual_seed(seed)
    # Unpack the values in x_range
    x_min, x_max = x_range
    # Sample sample_size points from a uniform distribution
    X = torch.rand(sample_size)
    # Get min and max
    min_x_sampled = torch.min(X)
    max_x_sampled = torch.max(X)
    # Rescale between x_min and x_max
    X = X * (x_max - x_min) + x_min
    # Compute hat(y)
    y_hat = X * w_star + b_star
    # Compute y (Add Gaussian noise)
    y = y_hat + torch.normal(torch.zeros(sample_size),
        sigma*torch.ones(sample_size))
    return X, y
```

Generate train & validation points

```
num_samples_train = 160
num_samples_validation = 20
num_samples_test = 20

# Set the seed
seed_train = 0
seed_validation = 1
seed_test = 2

# Set a value of noise (=sigma)
sigma = 1.3

# Define x_range
x = (-2, 2)

# Generate train data
X_train, y_train = create_dataset(
    sample_size=num_samples_train, sigma=sigma, w_star=w_star,
    b_star = b_star, x_range=x, seed=seed_train)

# Generate the validation data form the same distribution but with a different seed
X_val, y_val = create_dataset(
    sample_size=num_samples_validation, sigma=sigma, w_star=w_star,
    b_star = b_star, x_range=x, seed=seed_validation)
```

Visualize the data i

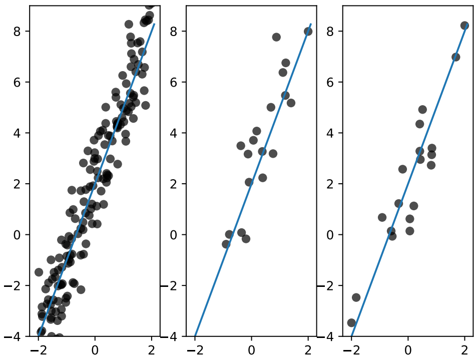
Usually, plots are done with Matplotlib or Seaborn

```
fig, ax = plt.subplots(1, 3) # Create a subplot

# You can use torch directly with matplotlib
ax[0].plot(X_train, y_train, 'ko', alpha = 0.7)
ax[1].plot(X_val, y_val, 'ko', alpha = 0.7)
ax[2].plot(X_test, y_test, 'ko', alpha = 0.7)

# Just to have some values in the x-axis
x_range = torch.arange(start=min(x) - 0.1, end=max(x) + 0.1, step=0.01)
for i in range(3):
    ax[i].set_ylim([-4, 9]) # You can see that these limits provide a better visualization
    ax[i].plot(x_range, w_star * x_range + b_star)
```

Visualize the data ii



Linear regression

- We need a linear **model** → already implemented in PyTorch : `torch.nn.Linear`
- We need a Loss Function. Regression → Mean Square Error (MSE). `torch.nn.MSELoss`
- We need an optimizer, let's stick to stochastic gradient descent, already implemented in PyTorch: `torch.optim.SGD`
- Last but not least: we want to use GPUs!

```
DEVICE = torch.device('cuda' if torch.cuda.is_available() else 'mps'
    if torch.backends.mps.is_available() else 'cpu')
```

Linear regression

- We then create the model, defining the loss function and optimizer
- First hyperparameter to tune: learning rate

```
model = nn.Linear(1, 1) # Dimension of input: 1, dimension of output: 1
loss_fn = nn.MSELoss()
learning_rate = 0.5
optimizer = optim.SGD(model.parameters(), lr=learning_rate)
```

What's happening at the beginning?

```
print("Initial w:", model.weight, "Initial b:\n", model.bias)
print("Value in x = 1:", model(torch.tensor([1.]))) # This computes w * 1 + b
print("Actual value in x = 1:", w_star * 1 + b_star)
# Loss evaluation
initial_loss_function = loss_fn(X_train.reshape(-1, 1), y_train.reshape(-1, 1))
print("Initial loss function:", initial_loss_function)
```

```
Initial w: Parameter containing:
tensor([[-0.2032]], requires_grad=True) Initial b:
 Parameter containing:
tensor([0.5817], requires_grad=True)
Value in x = 1: tensor([0.3785], grad_fn=<ViewBackward0>)
Actual value in x = 1: 5
Initial loss function: tensor(10.7261)
```

Training

We are almost ready for the training, but first of all, we have to put everything as needed for PyTorch

- Everything on GPU if we want to work on GPU
- Everything of the suitable type/shape

important put data in suitable dimensions

```
X_train = X_train.reshape(-1, 1).to(DEVICE)
y_train = y_train.reshape(-1, 1).to(DEVICE)
X_val = X_val.reshape(-1, 1).to(DEVICE)
y_val = y_val.reshape(-1, 1).to(DEVICE)
model = model.to(DEVICE)
```

The training loop

```python
n_steps = 10 # Number of updates of the gradient
for step in range(n_steps):
    model.train() # Set the model in training mode
    # Set the gradient to 0
    optimizer.zero_grad() # Or model.zero_grad()
    # Compute the output of the model
    y_hat = model(X_train)
    # Compute the loss
    loss = loss_fn(y_hat, y_train)
    # Compute the gradient
    loss.backward()
    # Update the parameters
    optimizer.step()
```

gradients are accumulated, it's important to set optimizer.zero.grad() -> so important

Evaluation process

Check what happens in the validation data

```python
for step in range(n_steps):
    # ###################
    # Code of previous slide here
    # ###################
    with torch.no_grad(): #
        # Compute the output of the model
        y_hat_val = model(X_val)
        # Compute the loss
        loss_val = loss_fn(y_hat_val, y_val)
        # At every step, print the losses
        print("Step:", step, "- Loss eval:", loss_val.item())
```

loss function \infty, it's not a number.

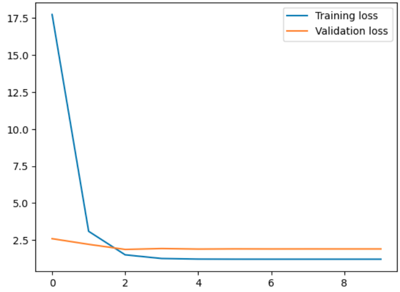
Training done!

Recall that we chose $w^* = 3$, $b^* = 2$.

```python
print("Training done, with an evaluation loss of {}".format(loss_val.item()))
# Get the final value of the parameters
print("Final w:", model.weight, "Final b:\n", model.bias)
```

u should see

something that decreases.




```
Training done, with an evaluation loss of 1.8968610763549805
Final w: Parameter containing:
tensor([[3.1482]], requires_grad=True) Final b:
 Parameter containing:
tensor([1.9967], requires_grad=True)
```

Linear regression is an old story tough...

... and hence is implemented in *many libraries*. Consider Scikit-Learn

-> use the library that they asked.

```
from sklearn.linear_model import LinearRegression
reg = LinearRegression().fit(X_train.to('cpu').numpy(),
y_train.to('cpu').numpy())
print("Final w:", reg.coef_, "Final b:\n", reg.intercept_)
```

```
Final w: [[3.1484804]] Final b:
 [1.9966233]
```

Some concluding remarks & general hints (from my experience)

- If a dedicated software/program does something very specific, it probably is better than a general-purpose program in doing that thing
- **However** from an educational perspective, good to test highly complicated routines on simple examples
- This lecture is fundamental for Assignment 1

Assignment 1 Presentation

....