

AI505/AI801, Optimization – Exercise Sheet 07

2026-04-13

i Solutions included.

Exercises with the symbol $+$ are to be done at home before the class. Exercises with the symbol $*$ will be tackled in class. The remaining exercises are left for self training after the exercise class. Some exercises are from the text book and the number is reported. They have the solution at the end of the book.

Exercise 1 $+$

Write the update rule for stochastic gradient with mini-batches of size m on a generical machine learning model $y = h(x)$ and with loss function ℓ .

Write the update formula with momentum in the case of mini-batch of size m .

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \alpha_k \frac{1}{m} \sum_{i=1}^m \nabla_{\boldsymbol{\theta}} \ell(\mathbf{x}_i, y_i, \boldsymbol{\theta})$$

with momentum:

$$\mathbf{v}_{k+1} = \beta \mathbf{v}_k - \alpha_k \frac{1}{m} \sum_{i=1}^m \nabla_{\boldsymbol{\theta}} \ell(\mathbf{x}_i, y_i, \boldsymbol{\theta})$$

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \mathbf{v}_{k+1}$$

Exercise 2 $*$

In a regression task we assume $h(\mathbf{x}; \mathbf{w}) = w_0 + w_1 x_1 + w_2 x_2 + \dots + w_d x_d$. For the estimation of the parameters $\mathbf{w} \in \mathbb{R}^{d+1}$ using the examples $\{(\mathbf{x}_1, y_1), \dots, (\mathbf{x}_n, y_n)\}$ we can use the least squares loss function:

$$\min R_n(\mathbf{w}) = \sum_{i=1}^n (h(\mathbf{x}_i; \mathbf{w}), y_i) = \|\mathbf{y} - X\mathbf{w}\|_2^2$$

where

$$X = \begin{bmatrix} 1 & x_{11} & x_{21} & \dots & x_{d1} \\ 1 & x_{11} & x_{21} & \dots & x_{d1} \\ \vdots & \ddots & & & \\ 1 & x_{1n} & x_{2n} & \dots & x_{dn} \end{bmatrix}$$

This problem admits a closed form solution by means of the normal equations $\mathbf{w} = (X^T X)^{-1} X^T \mathbf{y}$. You find the derivation of this result in these [slides from DM579/AI511](#).

The L_2 Regularized risk is

$$\min_{\mathbf{w}} R_n(\mathbf{w}) + \lambda \|\mathbf{w}\|_2^2 = \sum_{i=1}^n L(h(\mathbf{x}_i; \mathbf{w}), y_i) + \lambda \sum_{j=0}^d w_j^2 = \|\mathbf{y} - X\mathbf{w}\|_2^2 + \lambda \|\mathbf{w}\|_2^2$$

admits also a closed-form solution: $\mathbf{w} = (X^T X + \lambda I)^{-1} (X^T \mathbf{y})$.

Provide a computational analysis of the cost of computing the estimates of \mathbf{w} by means of these closed-form solutions and compare these costs with the cost of carrying out the gradient descent. When is the gradient descent faster?

We first note that the objective

$$f(w) = \|y - Xw\|_2^2$$

is always a convex quadratic function of w , since

$$\nabla^2 f(w) = 2X^T X \succeq 0.$$

Hence the problem is never nonconvex. If X has full column rank, then $X^T X \succ 0$, so the objective is strongly convex and the minimizer is unique. If X is rank-deficient, then the problem remains convex, but it is not strongly convex and the minimizer need not be unique.

For ridge regression,

$$f_\lambda(w) = \|y - Xw\|_2^2 + \lambda \|w\|_2^2, \quad \lambda > 0,$$

the Hessian is

$$\nabla^2 f_\lambda(w) = 2(X^T X + \lambda I) \succ 0,$$

so the problem is always strongly convex and has a unique minimizer.

For the asymptotic analysis, we ignore the difference between d and $d+1$, since it does not affect the big- O notation. Thus we take $X \in \mathbb{R}^{n \times d}$, where n is the number of data points and d is the feature dimension.

Exercise 0.a Closed-form least squares

The normal equations give

$$w^* = (X^T X)^{-1} X^T y.$$

The cost of computing this solution is:

- form $X^T X$: $O(nd^2)$,
- form $X^T y$: $O(nd)$,
- solve the $d \times d$ linear system: $O(d^3)$.

Hence the total cost is

$$O(nd^2 + d^3).$$

The same asymptotic cost applies to ridge regression,

$$w^* = (X^T X + \lambda I)^{-1} X^T y,$$

since adding λI costs only $O(d)$, which is negligible.

Exercise 0.b Remark on solving vs inverting

In practice, one should **solve** the linear system

$$(X^T X)w = X^T y$$

or

$$(X^T X + \lambda I)w = X^T y$$

rather than explicitly computing the inverse. Both approaches have the same asymptotic complexity $O(d^3)$, but solving is numerically more stable and has smaller constant factors. If the matrix is symmetric positive definite, Cholesky factorization is a natural choice.

Exercise 0.c Gradient descent for least squares

The gradient is

$$\nabla f(w) = 2X^\top(Xw - y).$$

A matrix-free gradient evaluation costs:

- compute Xw : $O(nd)$,
- compute $Xw - y$: $O(n)$,
- compute $X^\top(Xw - y)$: $O(nd)$.

Hence one full gradient step costs

$$O(nd).$$

If gradient descent runs for T iterations, the total cost is

$$O(Tnd).$$

For ridge regression,

$$\nabla f_\lambda(w) = 2X^\top(Xw - y) + 2\lambda w,$$

and the extra term $2\lambda w$ costs only $O(d)$, so the per-iteration complexity remains

$$O(nd).$$

Exercise 0.d When does gradient descent have linear convergence?

If X has full column rank, then the least-squares objective is strongly convex, and gradient descent converges linearly:

$$f(w_t) - f(w^*) \leq \left(1 - \frac{\mu}{L}\right)^t (f(w_0) - f(w^*)),$$

where

$$L = 2\lambda_{\max}(X^\top X), \quad \mu = 2\lambda_{\min}(X^\top X).$$

Thus the number of iterations needed to reach accuracy ε is

$$T = O\left(\kappa \log \frac{1}{\varepsilon}\right), \quad \kappa = \frac{L}{\mu} = \frac{\lambda_{\max}(X^\top X)}{\lambda_{\min}(X^\top X)}.$$

For ridge regression with $\lambda > 0$, the same linear convergence bound always holds, with condition number

$$\kappa_\lambda = \frac{\lambda_{\max}(X^\top X) + \lambda}{\lambda_{\min}(X^\top X) + \lambda}.$$

Hence the total cost of gradient descent for ridge is

$$O\left(nd \kappa_\lambda \log \frac{1}{\varepsilon}\right).$$

If $\lambda = 0$ and X is rank-deficient, then the problem is convex but not strongly convex. In that case, the standard strongly convex linear-convergence theorem does not apply, and the generic smooth convex rate is only sublinear:

$$f(w_t) - f(w^*) = O\left(\frac{1}{t}\right).$$

Equivalently, to achieve accuracy ε , one needs

$$T = O\left(\frac{1}{\varepsilon}\right)$$

iterations in the generic convex setting.

Exercise 0.e Comparison

The direct closed-form approach costs

$$O(nd^2 + d^3),$$

while gradient descent costs

$$O(Tnd).$$

Thus gradient descent is faster whenever

$$Tnd < nd^2 + d^3.$$

Dividing by nd , this becomes

$$T < d + \frac{d^2}{n}.$$

If the objective is strongly convex and we use the linear convergence bound

$$T = O\left(\kappa \log \frac{1}{\varepsilon}\right),$$

then gradient descent is faster whenever

$$\kappa \log \frac{1}{\varepsilon} < d + \frac{d^2}{n}.$$

In particular:

- if $n \gg d$, then nd^2 dominates d^3 , so the direct solve costs $O(nd^2)$, and gradient descent is faster when roughly

$$\kappa \log \frac{1}{\varepsilon} < d;$$

- if $d \gg n$, then d^3 dominates, so the direct solve costs $O(d^3)$, and gradient descent becomes much more attractive;
- if n and d are of the same order, then both terms are comparable and the direct solve costs $O(d^3)$, while one gradient step costs $O(d^2)$.

For background on convexity of least squares and ridge regression, and for the standard linear-convergence result of gradient descent on smooth strongly convex objectives, see Boyd and Vandenberghe, *Convex Optimization* (Chapter 9 on unconstrained minimization). An official PDF is available at https://web.stanford.edu/~boyd/cvxbook/bv_cvxbook.pdf. For additional background on least squares, see Boyd and Vandenberghe, *Introduction to Applied Linear Algebra: Vectors, Matrices, and Least Squares*, available at <https://web.stanford.edu/~boyd/vmls/vmls.pdf>.

Exercise 3

Consider now multiple logistic regression. In this case the hypothesis is that the probability of $y = 1$ given \mathbf{x} is given by

$$h(\mathbf{x}; \mathbf{w}) = p(y = 1 | \mathbf{x}; \mathbf{w}) = \frac{1}{1 + \exp(-(w_0 + w_1x_1 + \dots + w_dx_d))} = \frac{\exp(w_0 + w_1x_1 + \dots + w_dx_d)}{1 + \exp(w_0 + w_1x_1 + \dots + w_dx_d)}$$

The loss-likelihood function can be formulated as follows:

$$L = \prod_{i:y_i=1} p_i(\mathbf{x}_i) \prod_{i:y_i=0} (1 - p_i(\mathbf{x}_i))$$

$$\begin{aligned} R_n(\mathbf{w}) &= \log L(\mathbf{w}) = \\ &= \log \left(\prod_{i:y_i=1} \frac{\exp(w_0 + w_1 x_1 + \dots + w_d x_d)}{1 + \exp(w_0 + w_1 x_1 + \dots + w_d x_d)} \prod_{i:y_i=0} \frac{1}{1 + \exp(w_0 + w_1 x_1 + \dots + w_d x_d)} \right) \\ &= \sum_{i=1}^n y_i \log_e(p(\mathbf{x}_i)) + \sum_{i=1}^n (1 - y_i) \log_e(1 - p(\mathbf{x}_i)) = \\ &= \sum_{i=1}^n y_i (\mathbf{w}^T \mathbf{x}_i - \log(1 + \exp(\mathbf{w}^T \mathbf{x}_i))) + \sum_{i=1}^n (1 - y_i) (-\log(1 + \exp(\mathbf{w}^T \mathbf{x}_i))) \\ &= \sum_{i=1}^n y_i (\mathbf{w}^T \mathbf{x}_i) - \sum_{i=1}^n \log(1 + \exp(\mathbf{w}^T \mathbf{x}_i)) \end{aligned}$$

The minimization of the empirical risk: $\min R_n(\mathbf{x}; \mathbf{w})$ cannot be reformulated as a linear system in this case. Setting $\nabla R_n(\mathbf{w}) = 0$ gives a system of transcendental equations. But this objective function is convex and differentiable.

With some tedious manipulations, the gradient for logistic regression is

$$\nabla R_n(\mathbf{w}) = X^T \mathbf{s}$$

where vector \mathbf{s} has $s_i = -y_i h(-y_i \mathbf{w}^T \mathbf{x}_i)$ and h is the sigmoid function and

$$\nabla^2 R_n(\mathbf{w}) = X^T D X$$

where D is a diagonal matrix with

$$d_{ii} = h(y_i \mathbf{w}^T \mathbf{x}_i) h(-y_i \mathbf{w}^T \mathbf{x}_i)$$

It can be shown that $X^T D X$ is positive semidefinite and therefore logistic regression is convex (it becomes strictly convex if we add L_2 -regularization making the solution unique).

Hence, gradient descent converges to a global optimum. Alternately, another common approach is Newton's method. Requires computing Hessian $\nabla^2 R_n(\mathbf{w}_i)$.

What is the computational cost of gradient descent and of the Newton method for the logistic regression described?

Gradient descent costs $O(ndt)$.

Newton costs $O(nd^2 + d^3)$ per iteration to compute and invert $\nabla^2 R_n(\mathbf{w}_k)$. Newton typically requires substantially fewer iterations (1 if function is strongly convex). But for datasets with very large d , gradient descent might be faster. If $t < \max\{d, d^2/n\}$ then we should use the "slow" algorithm with fast iterations.

Exercise 4 +

Learn the basics of PyTorch <https://pytorch.org/tutorials/beginner/basics/intro.html>.

Exercise 5

Implement the logistic regression in pytorch using the MNIST dataset of handwritten number images. Use a model with 10 output nodes each implementing a sigmoid activation function.

Experiment with different versions of stochastic gradient: basic, mini-batch and batch. Compare stochastic gradient with other algorithms like Adam. You find a starting implementation in the appendix of this document.

Appendix

```
import torch
from torch import nn
from torch.utils.data import DataLoader
from torchvision import datasets
from torchvision.transforms import ToTensor
import matplotlib.pyplot as plt
import numpy as np

# Download training data from open datasets.
training_data = datasets.MNIST(root='./data',
                               train=True,
                               transform=ToTensor(),
                               download=True)

# Downloading test data
test_data = datasets.MNIST(root='./data',
                           train=False,
                           download=True,
                           transform=ToTensor())

print("number of training samples: " + str(len(training_data)) + "\n" +
      "number of testing samples: " + str(len(test_data)))

print("datatype of the 1st training sample: ", training_data[0][0].type())
print("size of the 1st training sample: ", training_data[0][0].size())

batch_size = 64

# Create data loaders.
train_dataloader = DataLoader(training_data, batch_size=batch_size)
test_dataloader = DataLoader(test_data, batch_size=batch_size)

for X, y in test_dataloader:
    print(f"Shape of X [N, C, H, W]: {X.shape}")
    print(f"Shape of y: {y.shape} {y.dtype}")
    break

device = torch.accelerator.current_accelerator().type if torch.accelerator.is_available() else "cpu"
print(f"Using {device} device")
```

```

# build custom module for logistic regression
# This model will take a -pixel image of handwritten digits as input and classify them into one of t
class LogisticRegression(torch.nn.Module):
    # build the constructor
    def __init__(self):
        super(LogisticRegression, self).__init__()
        self.linear = torch.nn.Linear(28*28, 10)

    # make predictions
    def forward(self, x):
        y_pred = torch.sigmoid(self.linear(x))
        return y_pred

model = LogisticRegression().to(device)
print(model)

# Optimizing the Model Parameters
loss_fn = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=1e-3)

def train(dataloader, model, loss_fn, optimizer):
    size = len(dataloader.dataset)
    model.train()
    losses=[]
    for batch, (X, y) in enumerate(dataloader):
        X, y = X.to(device), y.to(device)

        # Compute prediction error
        pred = model(X.view(-1,28*28))
        loss = loss_fn(pred, y)

        # Backpropagation
        loss.backward()
        optimizer.step()
        optimizer.zero_grad()

        losses.append(loss.item())
        if batch % 100 == 0:
            loss, current = loss.item(), (batch + 1) * len(X.view(-1,28*28))
            print(f"loss: {loss:>7f}  [{current:>5d}/{size:>5d}]")

    return losses

def test(dataloader, model, loss_fn):
    size = len(dataloader.dataset)
    num_batches = len(dataloader)
    model.eval()
    losses=[]
    test_loss, correct = 0, 0
    with torch.no_grad():
        for X, y in dataloader:
            X, y = X.to(device), y.to(device)

```

```

        pred = model(X.view(-1,28*28))
        test_loss += loss_fn(pred, y).item()
        losses.append(test_loss)
        correct += (pred.argmax(1) == y).type(torch.float).sum().item()
    test_loss /= num_batches
    correct /= size
    print(f"Test Error: \n Accuracy: {(100*correct):>0.1f}%, Avg loss: {test_loss:>8f} \n")
    return losses

epochs = 6
training_losses=[]
test_losses=[]

for t in range(epochs):
    print(f"Epoch {t+1}\n-----")
    training_losses += train(train_dataloader, model, loss_fn, optimizer)
    test_losses += test(test_dataloader, model, loss_fn)
print("Done!")
plt.plot(range(epochs), training_losses)
plt.plot( training_losses)
plt.plot(range(epochs), test_losses)
interval = int(np.ceil(len(training_data)/batch_size))
plt.xticks(range(1,interval*epochs+1,interval), range(1,epochs+1))
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.title('Training Loss Over Epochs')
plt.show()

# Saving Models
torch.save(model.state_dict(), "model.pth")
print("Saved PyTorch Model State to model.pth")

# Loading Models
model = LogisticRegression().to(device)
model.load_state_dict(torch.load("model.pth", weights_only=True))

classes = [
    "T-shirt/top",
    "Trouser",
    "Pullover",
    "Dress",
    "Coat",
    "Sandal",
    "Shirt",
    "Sneaker",
    "Bag",
    "Ankle boot",
]

model.eval()
x, y = test_data[0][0], test_data[0][1]
with torch.no_grad():

```

```
x = x.to(device)
pred = model(x.flatten())
```