Department of Mathematics and Computer Science
University of Southern Denmark, Odense

March 27, 2025
Marco Chiarandini

AI505 – Optimization

Sheet 05, Spring 2025

Solution:

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$^+$

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

Exercise 2$^+$

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.

Solution:

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

with momentum:

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

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

Exercise 3*

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

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

where

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

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

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

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 thegradient descent faster?

Solution:

The calculation of the solution via normal equations costs $O(nd^2 + d^3)$, matrix by matrix multiplication plus an inverse calculation. This is fine for $d = 5000$, but may be too slow for $d = 1,000,000$.
For the gradient descent we need first to calculate the gradient of R_n. In matrix form:

$$\nabla R_n(\mathbf{w}) = \nabla_{\mathbf{w}}(\mathbf{y} - X\mathbf{w})^T(\mathbf{y} - X\mathbf{w}) + \lambda I\mathbf{w}^2$$

$$\mathbf{w}_{k+1} = \mathbf{w}_k - \alpha_k \nabla R_n(\mathbf{w}) = \mathbf{w}_k - \alpha_k(X^T(X\mathbf{w}_k - y) + \lambda \mathbf{w}_k)$$

which costs $O(ndt)$, t as total number of iterations, and k as iteration number. Gradient descent is faster if t is not too big: If we only need $t < \max\{d, d^2/n\}$ iterations.

Iteration Complexity How many iterations does gradient descent need?
We are interested in reaching $\left\|\nabla f(\mathbf{w}_k)\right\|^2 = 0$ or more practical $\left\|\nabla f(\mathbf{w}_k)\right\|^2 \leq \epsilon$.
Give an ϵ how many iterations does it take for this to happen?

Theorem 1 *If we run for t iterations, we'll find at least one k with $\left\|\nabla f(\mathbf{w}_k)\right\|^2 = O(1/t)$.*

Theorem 2 *For functions that are bounded below and have a Lipschitz-continuous gradient, Gradient Descent requires $t = O(1/\epsilon)$ iterations to achieve $\left\|\nabla f(\mathbf{w}_k)\right\|^2 \leq \epsilon$. (note: dimension independent)*

So if computing gradient costs $O(nd)$, total cost of gradient descent is $O(nd/\epsilon)$, that is, $O(nd)$ per iteration for $O(1/\epsilon)$ iterations.
There is a dimension d beyond which gradient descent is faster than normal equations. In practice gradient descent converges much faster.

Theorem 3 *For convex functions we need $O(1/\epsilon)$ iterations to get ϵ-close to global optimum. We can get to $O(1/\sqrt{\epsilon})$.*

Is $O(1/\epsilon)$ a good iteration complexity?
Let's try to gain an idea about it. Recall that ϵ specifies the desired accuracy. The smaller the better.

ϵ	$1/\epsilon$	$\log(1/\epsilon)$	$t = O(1/\epsilon)$
10^{-1}	10	1	10
10^{-2}	100	2	100
10^{-3}	1000	3	1000
10^{-4}	10000	4	10000

We can think of $\log(1/\epsilon)$ as the desired "number of digits of accuracy". We want iteration complexity to grow slowly with $1/\epsilon$ and with the number of digits of accuracy. Instead, it grows exponentially with respect to this latter.

For t total number of iterations:

	iterations for error to go below ϵ	rate of convergence
error diminishing at rate $O(1/t)$	needs $O(1/\epsilon)$ iterations	sublinear arte
error diminishing at rate $O(\rho^t)$	needs $O(\log(1/\epsilon))$ iterations	linear rate
error diminishing at rate $O(\rho^{2^t})$	needs $O(\log\log(1/\epsilon))$ iterations	superlinear rate

Exercise 4

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

$$h(\mathbf{x};\mathbf{w}) = p(y = 1 \mid \mathbf{x};\mathbf{w}) = \frac{1}{1+\exp\left(-(w_0 + w_1x_1 + \ldots + w_dx_d)\right)} = \frac{\exp\left(w_0 + w_1x_1 + \ldots + w_dx_d\right)}{1+\exp\left(w_0 + w_1x_1 + \ldots + w_dx_d\right)}$$

The loss-likelihood function can be formulated as follows:

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

$$\begin{aligned}
R_n(\mathbf{w}) &= \log L(\mathbf{w}) = \\
&= \log\left(\prod_{i:y_i=1} \frac{\exp\left(w_0 + w_1x_1 + \ldots + w_dx_d\right)}{1+\exp\left(w_0 + w_1x_1 + \ldots + w_dx_d\right)} \prod_{i:y_i=0} \frac{1}{1+\exp\left(w_0 + w_1x_1 + \ldots + w_dx_d\right)}\right) \\
&= \sum_{i=1}^{n} y_i \log_e(p(\mathbf{x}_i)) + \sum_{i=1}^{n}(1-y_i)\log_e(1-p(\mathbf{x}_i)) = \\
&= \sum_{i=1}^{n} y_i(\mathbf{w}^T\mathbf{x}_i - \log\left(1+\exp\left(\mathbf{w}^T\mathbf{x}_i\right)\right) + \sum_{i=1}^{n}(1-y_i)(-\log\left(1+\exp\left(\mathbf{w}^T\mathbf{x}_i\right)\right) \\
&= \sum_{i=1}^{n} y_i(\mathbf{w}^T\mathbf{x}_i) - \sum_{i=1}^{n} \log\left(1+\exp\left(\mathbf{w}^T\mathbf{x}_i\right)\right)
\end{aligned}$$

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

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

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^TDX 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?

Solution:

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 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.

Exercise 6*

Recall that a way to measure rate of convergence is by the *limit of the ratio of successive errors*,

$$\lim_{k\to\infty} \frac{f(\mathbf{w}_{k+1}) - f(\mathbf{w}^*)}{f(\mathbf{w}_k) - f(\mathbf{w}^*)} = r$$

Different r values of give us different rates of convergence:

- If $r = 1$ we call it a sublinear rate.
- If $r \in (0, 1)$ we call it a linear rate.
- If $r = 0$ we call it a superlinear rate.

Consider the following sequences, which represent the error $e_k = \left\|F(\mathbf{w}_k) - F^*\right\|$ at iteration k of an optimization algorithm:

1. $e_k = 0.5^k$
2. $e_k = \frac{1}{k+1}$
3. $e_k = 0.1^k$
4. $e_k = \frac{1}{(k+1)^2}$
5. $e_k = \frac{1}{2^{2^k}}$

Tasks:

a) Classify the convergence rate of each sequence as linear, sublinear, superlinear, or quadratic.

b) Provide a justification for each classification by computing the ratio e_{k+1}/e_k or by using the definition of order of convergence.

Solution:

We analyze each sequence by computing the ratio e_{k+1}/e_k and checking how it behaves as $k \to \infty$.

- $e_k = 0.5^k$

  $$\frac{e_{k+1}}{e_k} = \frac{0.5^{k+1}}{0.5} = 0.5$$

  Since this ratio is a constant $r = 0.5$ with $0 < r < 1$, the convergence is linear.

- $e_k = \frac{1}{k+1}$

  $$\frac{e_{k+1}}{e_k} = \frac{1}{k+2} / \frac{1}{k+1} = \frac{k+1}{k+2}$$

  Taking the limit:

  $$\lim_{k\to\infty} \frac{k+1}{k+2} = 1$$

  Since the ratio tends to 1, the convergence is sublinear (very slow).

- $e_k = 0.1^k$

  $$\frac{e_{k+1}}{e_k} = \frac{0.1^{k+1}}{0.1^k} = 0.1$$

  Since this ratio is a constant $r = 0.1$ with $0 < r < 1$, the convergence is linear, but it is faster than Sequence 1 (since $r = 0.1$ is smaller than $r = 0.5$ and closer to 0 which is the superlinear rate).

- $e_k = \frac{1}{(k+1)^2}$

$$\frac{e_{k+1}}{e_k} = \frac{1}{(k+2)^2} / \frac{1}{(k+1)^2} = \frac{(k+1)^2}{(k+2)^2}$$

Taking the limit:

$$\lim_{k\to\infty} \frac{(k+1)^2}{(k+2)^2} = \left(\frac{k+1}{k+2}\right)^2 = 1$$

This suggests linear convergence.

- $e_k = \frac{1}{2^{2^k}}$

$$\lim_{k\to\infty} \frac{e_{k+1}}{e_k} = 0$$

This suggests superlinear convergence. However, to verify if it is faster than linear, we check:

$$\lim_{k\to\infty} \frac{e_{k+1}}{e_k^2} = \infty$$

Since this limit is not finite, the sequence is superlinear but not quadratic.

Exercise 7*

Consider applying gradient descent to the one-dimensional quadratic function

$$f(x) = \frac{1}{2}x^2$$

with the update rule:

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \alpha \nabla f(\mathbf{x}_k).$$

where $\nabla f(x) = x$.
Tasks:

a) Derive the update formula for $\mathbf{x}_k$.

b) Show that the error $e_k = \|\mathbf{x}_k\|$ follows an exponential decay when $0 < \alpha < 2$.

c) Compute $\frac{e_{k+1}}{e_k}$ and determine the rate of convergence for different values of α.

d) Set up a Python experiment where gradient descent is applied with different step sizes (α) and verify the theoretical convergence rate numerically.

Hint: Try $\alpha = 0.1, 0.5, 1, 1.5$ and observe how quickly the errors decrease.

Solution:

The update rule is

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \alpha \mathbf{x}_k = (1-\alpha)\mathbf{x}_k$$

Expanding the recursion:

$$x_1 = (1-\alpha)x_0$$
$$x_2 = (1-\alpha)x_1 = (1-\alpha)^2 x_0$$
$$x_3 = (1-\alpha)x_2 = (1-\alpha)^3 x_0.$$

By induction, the general formula is:

$$x_k = (1-\alpha)^k x_0.$$

The error is defined as $e_k = \|x_k\|$.
Since $x_k = (1-\alpha)^k x_0$, we have:

$$e_k = |(1-\alpha)^k||x_0|$$

Since $0 < \alpha < 2$, the term $|1-\alpha|$ satisfies:

$$|1-\alpha| < 1$$

Thus, as $k \to \infty$, we see exponential decay:

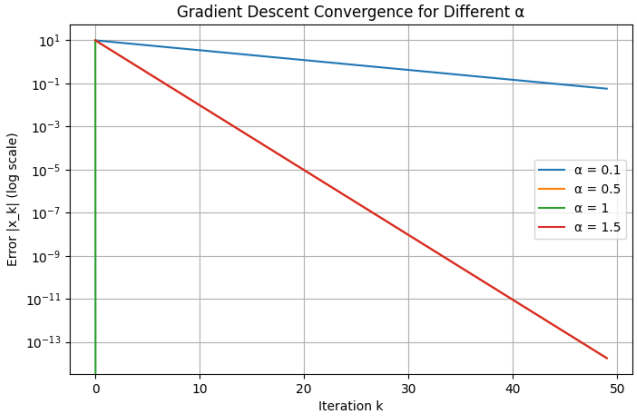
$$e_k = |1 - \alpha|^k e_0.$$

c) This result together with the analysis in the previous exercise confirms that gradient descent converges linearly in this simple quadratic case.
d)

```
import numpy as np
import matplotlib.pyplot as plt

# Function and gradient
def f(x):
    return 0.5 * x**2

def grad_f(x):
    return x # Since f(x) = (1/2) x^2, its gradient is simply x

# Gradient Descent Function
def gradient_descent(x0, alpha, num_iters):
    errors = []
    x_k = x0
    for _ in range(num_iters):
        errors.append(abs(x_k)) # Store absolute error |x_k|
        x_k = x_k - alpha * grad_f(x_k) # Gradient descent update
    return errors

# Initial point
x0 = 10 # Starting point

# Step sizes to test
alphas = [0.1, 0.5, 1, 1.5]

# Number of iterations
num_iters = 50

# Run gradient descent for different values of alpha
plt.figure(figsize=(8, 5))

for alpha in alphas:
    errors = gradient_descent(x0, alpha, num_iters)
    plt.semilogy(errors, label=f"\alpha = {alpha}")

# Plot settings
plt.xlabel("Iteration k")
plt.ylabel("Error |x_k| (log scale)")
plt.title("Gradient Descent Convergence for Different alpha")
plt.legend()
plt.grid()
#plt.show()
plt.savefig("convergence.png")
```

Explanation:

- We initialize $x_0 = 10$ (arbitrary nonzero starting point).
- For each α, we perform gradient descent for 50 iterations.
- We plot e_k on a semilog scale, where exponential decay appears as a straight line.
- Expected behavior:
  - If $0 < \alpha < 2$, the error decreases exponentially.
  Higher α (closer to 2) should converge faster but risks instability if $\alpha \geq 2$.



Expected Observations:

- For small α (e.g., 0.1): Convergence is slow.
- For moderate α (e.g., 0.5, 1): Faster convergence.
- For large α (e.g., 1.5): Convergence is still fast but close to instability.
- If we set $\alpha = 2$, we would see oscillations instead of convergence.

The script numerically confirms the exponential decay of errors when $0 < \alpha < 2$.

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 the 10 output classes of digits 0 to 9.
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())
    predicted, actual = classes[pred[0].argmax(0)], classes[y]
    print(f'Predicted: "{predicted}", Actual: "{actual}"')
```