AI505/AI801, Optimization – Exercise Sheet 06

2026-03-22

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.

Exercise 2 *

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

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

where

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

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

The L_2 Regularized risk is

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

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

Provide a computational analysis of the cost of computing the estimates of $\boldsymbol{w}$ 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?

Exercise 3

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

$$h(\boldsymbol{x}; \boldsymbol{w}) = p(y = 1 \mid \boldsymbol{x}; \boldsymbol{w}) = \frac{1}{1 + \exp\left(-(w_0 + w_1 x_1 + ... + w_d x_d)\right)} = \frac{\exp\left(w_0 + w_1 x_1 + ... + w_d x_d\right)}{1 + \exp\left(w_0 + w_1 x_1 + ... + w_d x_d\right)}$$

The loss-likelihood function can be formulated as follows:

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

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

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

With some tedious manipulations, the gradient for logistic regression is

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

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

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

where D is a diagonal matrix with

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

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 $\nabla^2 R_n(\boldsymbol{w}_i)$.

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

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())
```