ECON 425 HW6 Solutions

February 19, 2024

Problem 1 (backpropagation and network layers)

Suppose x and y are scalars and consider a neural network with one neuron in each of the two hidden layers,

$$x \xrightarrow{w_0} h_1 \xrightarrow{w_1} h_2 \xrightarrow{w_2} \hat{y},$$

where the activation function for the hidden layers is the sigmoid $\sigma(z) = \frac{1}{1+e^{-z}}$ and the activation function for the output layer is the identity $\sigma_{out}(z) = z$. Assume that the bias parameters b_0, b_1, b_2 are all zeros.

(i) Derive the gradient of the prediction $\hat{y}$ w.r.t. the weights $w = (w_0, w_1, w_2)$. Show that the gradient w.r.t. w_k, where $k \in \{0, 1, 2\}$, only depends on outputs and weights of layers $\ell \geq k$ ($\ell = 0$ is the input layer). [This is what allows backpropagation in NNs.]

(ii) Consider a 1-layer NN obtained by removing one of the hidden layers from the 2-layer NN above. Suppose the true data generating process is $y = \sigma(x)$, where $x \sim N(0, 1)$. Generate $n = 1,000,000$ data points and fit both NNs by minimizing the average squared loss (you need not use backpropagation here; use scipy.optimize.minimize). Report training errors and optimized weights. Explain why in this case adding another layer *increases* the training error.

Solution The output of the network is

$$\hat{y} = w_2 h_2 = w_2 \sigma(w_1 h_1) = w_2 \sigma(w_1 \sigma(w_0 x)).$$

For the sigmoid function, its derivative is $\sigma'(z) = \sigma(z)(1 - \sigma(z))$. Using this, we obtain

$$\begin{aligned}
\frac{\partial \hat{y}}{\partial w_2} &= h_2, \\
\frac{\partial \hat{y}}{\partial w_1} &= w_2 h_2 (1 - h_2) h_1, \\
\frac{\partial \hat{y}}{\partial w_0} &= w_2 h_2 (1 - h_2) w_1 h_1 (1 - h_1) x.
\end{aligned}$$

[3]:
```
import numpy as np
from scipy.optimize import minimize


sigmoid = lambda x: 1 / (1 + np.exp(-x))

np.random.seed(3)
x = np.random.normal(0, 1, size=(1000000,1))
```

```python
y = sigmoid(x)

# 1-layer NN
emp_loss_1 = lambda w: np.mean((y - w[1]*sigmoid(w[0]*x))**2)
# 2-layer NN
emp_loss_2 = lambda w: np.mean((y - w[2]*sigmoid(w[1]*sigmoid(w[0]*x)))**2)

w_ini = [0, 0]
results_1 = minimize(emp_loss_1, w_ini)
print('========= 1-LAYER NN =========')
print(results_1)
w_ini = [0, 0, 0]
results_2 = minimize(emp_loss_2, w_ini)
print('========= 2-LAYER NN =========')
print(results_2)
```

```
========= 1-LAYER NN =========
  message: Optimization terminated successfully.
  success: True
   status: 0
      fun: 5.985852810098471e-13
        x: [ 1.000e+00  1.000e+00]
      nit: 10
      jac: [-2.373e-07 -2.607e-07]
 hess_inv: [[ 2.407e+01 -2.372e+00]
            [-2.372e+00  1.768e+00]]
     nfev: 33
     njev: 11
========= 2-LAYER NN =========
  message: Optimization terminated successfully.
  success: True
   status: 0
      fun: 0.010043226212701954
        x: [ 5.592e+00  3.454e+00  6.734e-01]
      nit: 50
      jac: [-2.119e-08 -1.860e-07 -7.501e-07]
 hess_inv: [[ 7.194e+03  1.054e+03 -6.835e+00]
            [ 1.054e+03  1.681e+03 -3.955e+01]
            [-6.835e+00 -3.955e+01  1.780e+00]]
     nfev: 252
     njev: 63
```

Problem 2 Use the dataset card_transdata.csv from the previous homework and maintain the same train-test split. For coding instructions, see *https://pytorch.org/tutorials/beginner/basics/optimization_tutorial.html*

Fit a feedforward neural network with two ReLU layers using stochastic gradient descent (SGD). Experiment with the number of neurons per layer, the number of epochs, the learning rate for SGD, and the batch size for backpropagation. Report accuracy and F1 score on the test sample. Does your model perform better than a simple decision tree from the last homework?

[6]:
```
import numpy as np
import pandas as pd
import torch
from torch import nn
import torch.optim as optim
from torch.utils.data import TensorDataset, DataLoader
from torchinfo import summary


# SET PARAMETERS HERE ---------
batch_size = 500
epochs = 10
learning_rate = 0.01
n_train = 500000

datapath = '/Users/franguri/Library/CloudStorage/Dropbox/_TEACHING_/Econ 425 ML␣
↪UCLA Winter 2024/WEEK 5 Imbalanced data/card_transdata.csv'
data = pd.read_csv(datapath, header=0)
x_all = data.drop('fraud', axis=1)
y_all = data['fraud']
x_train, y_train = x_all[:n_train], y_all[:n_train]
x_test, y_test = x_all[n_train:], y_all[n_train:]

class NNModel(nn.Module):
    def __init__(self , input_size):
        super(NNModel , self).__init__()
        self.flatten = nn.Flatten()
        self.sequential = nn.Sequential(
            nn.Linear(input_size , 10),
            nn.ReLU(),
            # nn.Dropout (0.4),
            nn.Linear(10, 1),
            nn.Sigmoid())

    def forward(self , x):
        x = self.flatten(x)
        return torch.flatten(self.sequential(x))


def train_loop(dataloader, model, loss_fn, optimizer):
    size = len(dataloader.dataset)
    # Set the model to training mode - important for batch normalization and␣
↪dropout layers
    # Unnecessary in this situation but added for best practices
    model.train()
```

```python
    for batch, (X, y) in enumerate(dataloader):
        # Compute prediction and loss
        pred = model(X)
        loss = loss_fn(pred, y)

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

        if batch % 100 == 0:
            loss, current = loss.item(), batch * batch_size + len(X)
            # print(f"loss: {loss:>7f}  [{current:>5d}/{size:>5d}]")


def test_loop(dataloader, model, loss_fn):
    # Set the model to evaluation mode - important for batch normalization and␣
 ↪dropout layers
    model.eval()
    size = len(dataloader.dataset)
    num_batches = len(dataloader)
    test_loss, correct, true_negs, true_poss, false_negs, false_poss = 0, 0, 0,␣
 ↪0, 0, 0

    # Evaluating the model with torch.no_grad() ensures that no gradients are␣
 ↪computed during test mode
    # also serves to reduce unnecessary gradient computations and memory usage␣
 ↪for tensors with requires_grad=True
    with torch.no_grad():
        for X, y in dataloader:
            pred = model(X)
            test_loss += loss_fn(pred, y).item()
            correct += ((pred>0.5) == y).type(torch.float).sum().item()
            false_negs = torch.logical_and(pred<=0.5, y==1).sum().item()
            false_poss = torch.logical_and(pred>0.5, y==0).sum().item()
            true_poss = torch.logical_and(pred>0.5, y==1).sum().item()

    test_loss /= num_batches
    correct /= size
    precision = true_poss / (true_poss + false_poss)
    recall = true_poss / (true_poss + false_negs)
    print(f"Accuracy: {(100*correct):>0.1f}%")
    print(f"Precision: {(100*precision):>0.1f}%")
    print(f"Recall: {(100*recall):>0.1f}%")


model = NNModel(x_train.shape[1])
```

```python
loss_fn = nn.CrossEntropyLoss()
optimizer = optim.SGD(model.parameters(), lr=learning_rate)
summary(model, input_size=x_train.shape, col_names=['input_size',␣
 ↪'output_size', 'num_params'])

data_train = TensorDataset(torch.tensor(x_train.values.astype(np.float32)),␣
 ↪torch.tensor(y_train.values.astype(np.float32)))
train_dataloader = DataLoader(data_train, batch_size=batch_size)

data_test = TensorDataset(torch.tensor(x_test.values.astype(np.float32)), torch.
 ↪tensor(y_test.values.astype(np.float32)))
test_dataloader = DataLoader(data_test, batch_size=batch_size)

for t in range(epochs):
    print(f"Epoch {t+1}\n-------------------------------")
    train_loop(train_dataloader, model, loss_fn, optimizer)
    test_loop(test_dataloader, model, loss_fn)
```

```
Epoch 1
-------------------------------
Accuracy: 89.4%
Precision: 51.2%
Recall: 95.3%
Epoch 2
-------------------------------
Accuracy: 63.9%
Precision: 18.8%
Recall: 100.0%
Epoch 3
-------------------------------
Accuracy: 67.0%
Precision: 20.5%
Recall: 100.0%
Epoch 4
-------------------------------
Accuracy: 78.0%
Precision: 28.6%
Recall: 97.7%
Epoch 5
-------------------------------
Accuracy: 92.5%
Precision: 59.2%
Recall: 97.7%
Epoch 6
-------------------------------
Accuracy: 81.3%
Precision: 35.2%
```

```
Recall: 100.0%
Epoch 7
-------------------------------
Accuracy: 68.7%
Precision: 21.3%
Recall: 100.0%
Epoch 8
-------------------------------
Accuracy: 88.9%
Precision: 48.3%
Recall: 100.0%
Epoch 9
-------------------------------
Accuracy: 93.4%
Precision: 60.9%
Recall: 97.7%
Epoch 10
-------------------------------
Accuracy: 90.6%
Precision: 52.4%
Recall: 100.0%
```

[46]:

```

```