Rohan Nyati

Roll No.-R177219148

SAP ID-500075940

Batch-5 AI&ML

Neural Networks Lab

Lab-5

Implement Optimal Brain Damage & Optimal Brain Surgery in your already designed network.

Optimal Brain Damage (OBD) and Optimal Brain Surgeon (OBS) represent two popular pruning procedures; however, pruning large networks trained on voluminous data sets using these methods easily becomes intractable. We present a number of approximations and discuss practical issues in real-world pruning, and use as an example a network trained to predict protein coding regions in DNA sequences. The efficiency of OBS on large networks is compared to OBD, and it turns out that OBD is preferable to OBS, since more weights can be removed using less computational effort.

Optimal Brain Surgeon (OBS) is significantly better than magnitude-based methods and Optimal Brain Damage, which often remove the wrong weights. OBS permits pruning of more weights than other methods (for the same error on the training set), and thus yields better generalization on test data.

Crucial to OBS is a recursion relation for calculating the inverse Hessian matrix H-' from training data and structural information of the net. OBS permits a 76%, a 62%, and a 90% reduction in weights over backpropagation with weight decay on three benchmark MONK'S problems. Of OBS, Optimal Brain Damage, and a magnitude-based method, only OBS deletes the correct weights from a trained XOR network in every case

Objective functions play a central role in this field; therefore it is more than reasonable to define the saliency of a parameter to be the change in the objective function caused by deleting that parameter

Hessian matrix is a matrix of second order partial derivatives

$$\mathbf{H}_f = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1\,\partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_1\,\partial x_n} \\ \frac{\partial^2 f}{\partial x_2\,\partial x_1} & \frac{\partial^2 f}{\partial x_2^2} & \cdots & \frac{\partial^2 f}{\partial x_2\,\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_n\,\partial x_1} & \frac{\partial^2 f}{\partial x_n\,\partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_n^2} \end{bmatrix},$$

The Optimal Brain Damage procedure can be carried out as follows:

1. Choose a reasonable network architecture
2. Train the network until a reasonable solution is obtained
3. Compute the second derivatives hu for each parameter 4.

   Compute the saliencies for each parameter:

$$s_k = h_{kk} u_k^2 / 2$$

5. Sort the parameters by saliency and delete some lowsaliency parameters
6. Iterate to step 2

Deleting a parameter is defined as setting it to 0 and freezing it there. Several variants of the procedure can be devised, such as decreasing the ... 41ues of the low saliency parameters instead of simply setting them to 0, or allowing the deleted parameters to adapt again after they have been set to 0

The Optimal Brain Surgery procedure can be carried out as follows:

1. Train a "reasonably large" network to mini- ~num error.
2. Compute H-'.
3. Find the q that gives the smallest saliency $L_q = u_q^2 / (2[\mathbf{H}^{-1}]qq)$. If this candidate error increase is nuclei smaller than E, then the qth weight should he deleted, and we proceed to step 4: otherwise go to step 5. (Other stopping criteria can be used too.)
4. Use the q from step 3 to update all weights (Eq. 5). Go to step 2.
5. No more weights can he deleted without large increase in E. (At this point it may be desirable to retrain the network.)

The method described here Optimal Brain Surgeon (OBS) - accepts the criterion makes no restrictive assumptions about the form of the network's Hessian. OBS thereby eliminates the correct weights. Moreover, unlike other methods, OBS does not demand (typically slow) retraining after the pruning of a weight.

Following is the model implemented in the last lab.

Neural Network Lab-5

February 22, 2022

Shreyansh Gupta Roll No.-R177219175 SAP ID-500076514 Batch-6 AI&ML

1 Import PyTorch

[2]:
```
import torch
import torchvision
import torchvision.datasets as datasets
import torch.nn as nn
import torchvision.transforms as transforms
import matplotlib.pyplot as plt
```

2 Initialize Hyper-parameters

[3]:
```
input_size=784
hidden_size=150
num_classes=15
num_epochs=5
batch_size=150
learning_rate=0.001
```

3 Build the Feedforward Neural Network

[139]:
```
class NeuralNet(nn.Module):
    def __init__(self, input_size, hidden_size, num_classes):
        super(NeuralNet, self).__init__()
        self.l1 = nn.Linear(input_size, hidden_size)
        self.relu = nn.ReLU()
        self.l2 = nn.Linear(hidden_size, num_classes)

    def forward(self, x):
        out = self.l1(x)
        out = self.relu(out)
        out = self.l2(out)
        return out
```

4 Instantiate the FNN

[140]:
```
model = NeuralNet(input_size, hidden_size, num_classes)
```

5 Enable GPU

[141]:
```
device= torch.device('cuda' if torch.cuda.is_available()
else 'cpu')
```

6 Choose the Loss Function and Optimizer

[142]:
```
criterion = nn.CrossEntropyLoss() optimizer =
torch.optim.Adam(model.parameters(), lr=learning_rate)
```

7 Training the FNN Model

[143]:
```
n_total_steps = len(train_loader)
for epoch in range(num_epochs):
    for i, (images, labels) in enumerate(train_loader):
        images = images.reshape(-1, 28*28).to(device)
        labels= labels.to(device)

        outputs = model(images)
        loss = criterion(outputs, labels)

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

        if (i+1) % 100 == 0:
                print (f'Epoch [{epoch+1}/{num_epochs}], step[{i+1}/
 ↪{n_total_steps}], loss: {loss.item():.4f}')
```

```
Epoch [1/5], step[100/400], loss: 0.3850
Epoch [1/5], step[200/400], loss: 0.3349
Epoch [1/5], step[300/400], loss: 0.1991
Epoch [1/5], step[400/400], loss: 0.2551
Epoch [2/5], step[100/400], loss: 0.2451
Epoch [2/5], step[200/400], loss: 0.2422
Epoch [2/5], step[300/400], loss: 0.2784
Epoch [2/5], step[400/400], loss: 0.2823
```

```
Epoch [3/5], step[100/400], loss: 0.1294
Epoch [3/5], step[200/400], loss: 0.1487
Epoch [3/5], step[300/400], loss: 0.0640
Epoch [3/5], step[400/400], loss: 0.1173
Epoch [4/5], step[100/400], loss: 0.1329
Epoch [4/5], step[200/400], loss: 0.1056
Epoch [4/5], step[300/400], loss: 0.1117
Epoch [4/5], step[400/400], loss: 0.1124
Epoch [5/5], step[100/400], loss: 0.1078
Epoch [5/5], step[200/400], loss: 0.0448
Epoch [5/5], step[300/400], loss: 0.0944
Epoch [5/5], step[400/400], loss: 0.0841
```

8 Testing the FNN Model

[144]:
```
with torch.no_grad():
    n_correct = 0
    n_samples = 0
    for images, labels in test_loader:
        images = images.reshape(-1, 28*28).to(device)
        labels = labels.to(device)
        outputs = model(images)
        _, predicted = torch.max(outputs.data, 1)
        n_samples += labels.size(0)
        n_correct += (predicted == labels).sum().item()
```

[ ]:
```

```