Optimal Brain Damage

Lynn Kelly Tchoufa

Introduction

Most successful applications of neural network learning to real-world problems have been achieved using highly structured networks of rather large size

Design tools and techniques for comparing different architectures and minimizing the network size will be needed.

We introduce a new technique called Optimal Brain Damage (OBD) for reducing the size of a learning network by selectively deleting weights.

Optimal Brain Damage

Optimal Brain Damage

The basic idea of OBD is that it is possible to take a perfectly reasonable network, delete half (or more) of the weights and wind up with a network that works just as well, or better.

-
- A simple strategy consists in deleting parameters with small "saliency", i.e. those whose deletion will have the least effect on the training error.
- Retrain the model
- Iterate
- Methode: Second order derivative

Procedure

The OBO 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 h_kk for each parameter

4. Compute the saliencies for each parameter: Sk = h_kku2/2

5. Sort the parameters by saliency and delete some low-saliency parameters

6. Iterate to step 2

The Experiment

The simulation results given were obtained using back-propagation applied to handwritten digit recognition.

It was trained on a database of segmented handwritten zip code digits and printed digits containing approximately 9300 training examples and 3350 test examples.

More details can be obtained from the companion paper (Le Cun et al., 1990b).

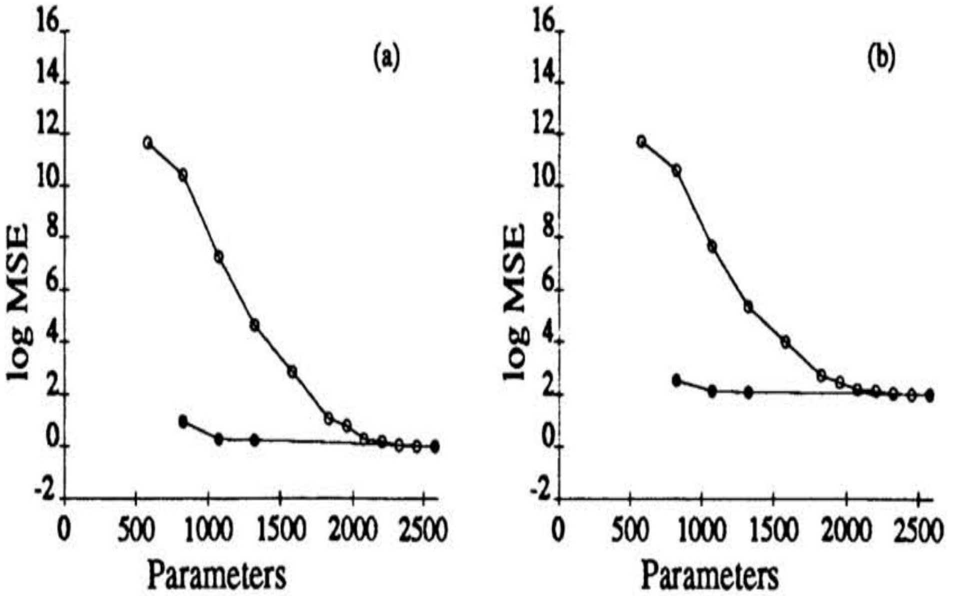


Figure 2: Objective function (in dB) versus number of parameters, without retraining (upper curve), and after retraining (lower curve). Curves are given for the training set (a) and the test set (b).

Implementation

The network architecture

```
class LeNet(nn.Module):
    def __init__(self):
        super(LeNet, self).__init__()

        ## cnn layers
        self.conv1 = nn.Sequential(
            nn.Conv2d(in_channels=1, out_channels=16, kernel_size=5, stride=1, padding=2),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2),
        )

        self.conv2 = nn.Sequential(
            nn.Conv2d(16, 32, 5, 1, 2),
            nn.ReLU(),
            nn.MaxPool2d(2),
        )
        # fully connected layer, output 10 classes
        self.out = nn.Linear(32 * 7 * 7, 10)

    def forward(self, x):
        x = self.conv1(x)
        x = self.conv2(x)

        # flatten the output of conv2 to (batch_size, 32 * 7 * 7)

        x = x.reshape(x.shape[0], -1)

        output = self.out(x)
        return output, x    # return x for visualization
```

Training results

The model was trained on MNIST dataset

Using:

- **Crossentropy loss**
- **Adam optimiser**
- **lr=0.01**
- **3 epochs**

```
/usr/local/lib/python3.7/dist-packages/torch/nn/functional.py:718: UserWarning: Named tensors and all their associated APIs are an experimental feature and subject to change. Please do not use them f
  return torch.max_pool2d(input, kernel_size, stride, padding, dilation, ceil_mode)
Train Epoch: 1 [0/60000 (0%)]	Loss: 2.308944
Train Epoch: 1 [10000/60000 (17%)]	Loss: 0.139430
Train Epoch: 1 [20000/60000 (33%)]	Loss: 0.155636
Train Epoch: 1 [30000/60000 (50%)]	Loss: 0.134818
Train Epoch: 1 [40000/60000 (67%)]	Loss: 0.144069
Train Epoch: 1 [50000/60000 (83%)]	Loss: 0.089038
/usr/local/lib/python3.7/dist-packages/torch/nn/_reduction.py:42: UserWarning: size_average and reduce args will be deprecated, please use reduction='sum' instead.
  warnings.warn(warning.format(ret))

Test set: Avg. loss: -9.2386, Accuracy: 9831/10000 (98%)

Train Epoch: 2 [0/60000 (0%)]	Loss: 0.055944
Train Epoch: 2 [10000/60000 (17%)]	Loss: 0.047586
Train Epoch: 2 [20000/60000 (33%)]	Loss: 0.057990
Train Epoch: 2 [30000/60000 (50%)]	Loss: 0.027995
Train Epoch: 2 [40000/60000 (67%)]	Loss: 0.074710
Train Epoch: 2 [50000/60000 (83%)]	Loss: 0.029778

Test set: Avg. loss: -11.8680, Accuracy: 9816/10000 (98%)

Train Epoch: 3 [0/60000 (0%)]	Loss: 0.031841
Train Epoch: 3 [10000/60000 (17%)]	Loss: 0.088782
Train Epoch: 3 [20000/60000 (33%)]	Loss: 0.029479
Train Epoch: 3 [30000/60000 (50%)]	Loss: 0.159364
Train Epoch: 3 [40000/60000 (67%)]	Loss: 0.002643
Train Epoch: 3 [50000/60000 (83%)]	Loss: 0.058338

Test set: Avg. loss: -13.7654, Accuracy: 9831/10000 (98%)
```

The next step is to

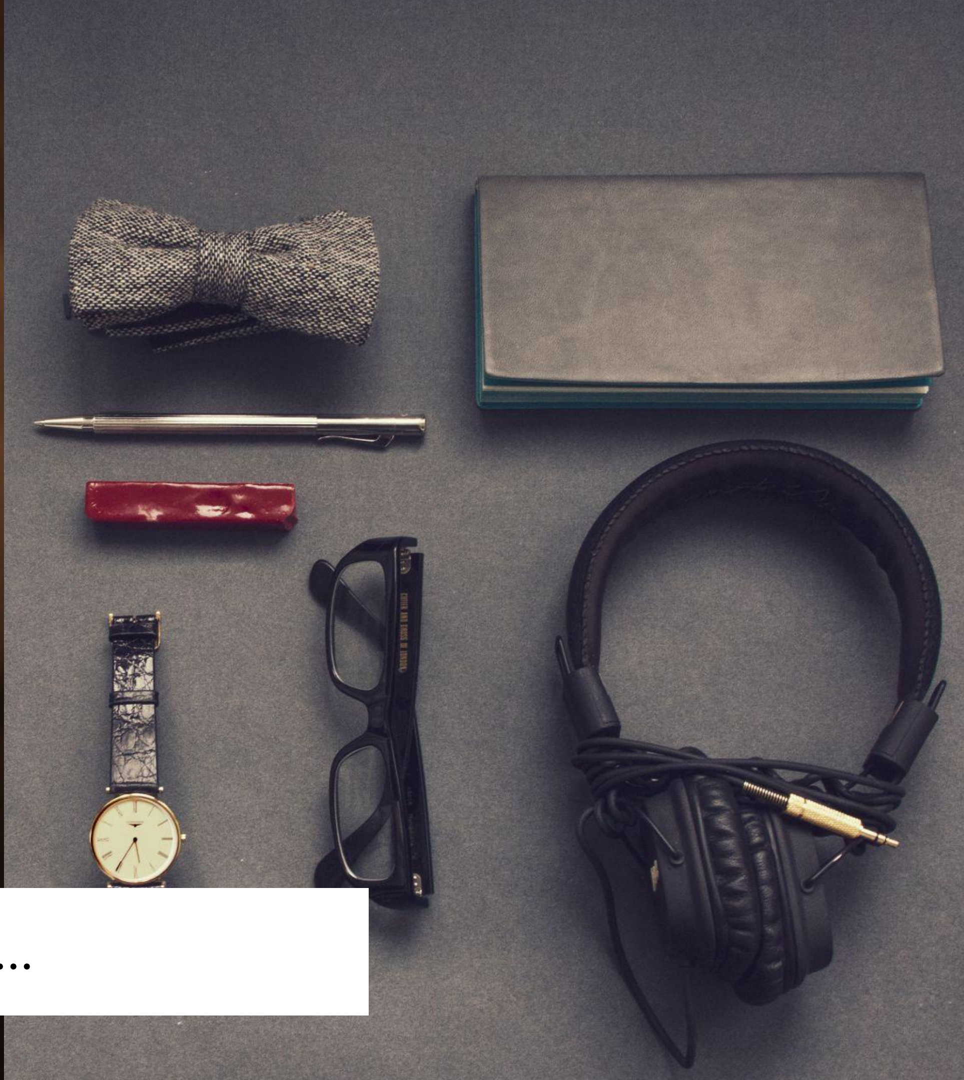
compute the Hessian matrix

Compute the saliencies

And retrain

```
def compute_hessian():

  for i, (images, labels) in enumerate(train_loader):

      images = images.to(device)
      labels = labels.to(device)
      # Forward pass
      outputs = loaded_model(images)

      for name, param in loaded_model.named_parameters():

        p = param
        loss = criterion(outputs[0], labels)
        print(loss)

        # Backward and optimize
        optimizer.zero_grad()
        loss.backward(retain_graph=True)
        grad_params = torch.autograd.grad(loss, p, create_graph=True,allow_unused=True)  # p is the weight matrix for a particular layer
        hess_params = torch.zeros_like(grad_params[0])


        for i in range(grad_params[0].size(0)):
            for j in range(grad_params[0].size(1)):

                hess_params[i, j] = torch.autograd.grad(grad_params[0][i][j], p, retain_graph=True)[0][i, j]

        optimizer.step()

        return hess_params
```

```
[ ] h=compute_hessian()

tensor(0.0379, device='cuda:0', grad_fn=<NllLossBackward>)
---------------------------------------------------------------------------
RuntimeError                              Traceback (most recent call last)
<ipython-input-20-535b03634c36> in <module>()
----> 1 h=compute_hessian()

2 frames

/usr/local/lib/python3.7/dist-packages/torch/autograd/__init__.py in _make_grads(outputs, grads)
     48             if out.requires_grad:
```



The Implementation results is yet to come...

Conclusion

- Optimal Brain Damage interactively can be used to reduce the number of parameters in a practical neural network by a factor of four.

- It Improves the network's speed improved significantly,

- and its recognition accuracy increased slightly.