Advanced Data Analysis
Homework Week 7

Aswin Vijay

June 12, 2023

1] Math

We are asked to derive $\frac{\partial J}{\partial x_i}$ where x_i is the $i'th$ units pre-activation value and J is the loss. We are given $\frac{\partial J}{\partial u_i}$, where u_i is the output after the batch normalization operation. We also have,

$$\mu_{\mathcal{B}} = \frac{1}{m}\sum_{i=1}^{m} x_i \quad mini-batchmean \tag{1}$$

$$\sigma_{\mathcal{B}}^2 = \frac{1}{m}\sum_{i=1}^{m} (x_i - \mu_\beta)^2 \quad mini-batch\ variance \tag{2}$$

$$\hat{x}_i = \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \quad normalized\ x_i \tag{3}$$

$$u_i = \gamma \hat{x}_i + \beta \quad Scale\ and\ shift \tag{4}$$

To derive $\frac{\partial J}{\partial x_i}$ we write out the partial derivatives using chain rule since $\hat{x}_i(x_i, \mu, \sigma^2)$, $\sigma^2(x_i, \mu)$ and $\mu(x_i)$, three variables are dependent on x_i so:

$$\frac{\partial J}{\partial x_i} = \frac{\partial J}{\partial \hat{x}_i} \cdot \frac{\partial \hat{x}_i}{\partial x_i} + \frac{\partial J}{\partial \mu} \cdot \frac{\partial \mu}{\partial x_i} + \frac{\partial J}{\partial \sigma^2} \cdot \frac{\partial \sigma^2}{\partial x_i}$$

The above derivatives are computed as follows,

$$\frac{\partial J}{\partial \hat{x}_i} = \frac{\partial J}{\partial u_i} \cdot \frac{\partial u_i}{\partial \hat{x}_i} = \gamma \cdot \frac{\partial J}{\partial u_i} \ From\ (4)$$

$$\frac{\partial \hat{x}_i}{\partial x_i} = \frac{1}{\sqrt{\sigma^2 + \epsilon}} \ From\ (3)$$

Computing J derivatives w.r.t μ,

$$\begin{aligned}
\frac{\partial J}{\partial \mu} &= \sum_{i=1}^{m} \frac{\partial J}{\partial \hat{x}_i} \cdot \frac{\partial \hat{x}_i}{\partial \mu} + \frac{\partial J}{\partial \sigma^2} \cdot \frac{\partial \sigma^2}{\partial \mu} \ \textit{Summing over batch} \\
\frac{\partial \hat{x}_i}{\partial \mu} &= \frac{-1}{\sqrt{\sigma^2+\epsilon}} \ \textit{From } (3) \\
\frac{\partial \sigma^2}{\partial \mu} &= \frac{-2}{m}\sum_{i=1}^{m}(x_i-\mu) = -2\left(\frac{1}{m}\sum_{i=1}^{m} x_i - \mu\right) = 0 \ \textit{From } (2,1)
\end{aligned}$$

Computing J derivatives w.r.t σ,

$$\begin{aligned}
\frac{\partial J}{\partial \sigma^2} &= \sum_{i=1}^{m} \frac{\partial J}{\partial \hat{x}_i} \cdot \frac{\partial \hat{x}_i}{\partial \sigma^2} \ \textit{Summing over batch} \\
\frac{\partial \hat{x}_i}{\partial \sigma^2} &= \frac{-1}{2}(\sigma^2+\epsilon)^{-\frac{3}{2}}(x_i-\mu) \ \textit{From } (3)
\end{aligned}$$

Putting everything together we have,

$$\begin{aligned}
\frac{\partial J}{\partial \mu} &= \sum_{i=1}^{m} \frac{\partial J}{\partial \hat{x}_i} \cdot \frac{-1}{\sqrt{\sigma^2+\epsilon}} + 0, \ \textit{then} \\
\frac{\partial J}{\partial x_i} &= \frac{\partial J}{\partial \hat{x}_i} \cdot \frac{1}{\sqrt{\sigma^2+\epsilon}} + \sum_{i=1}^{m} \frac{\partial J}{\partial \hat{x}_i} \cdot \frac{-1}{\sqrt{\sigma^2+\epsilon}} \cdot \frac{\partial \mu}{\partial x_i} + \frac{-1}{2}(\sigma^2+\epsilon)^{-\frac{3}{2}} \sum_{j=1}^{m} \frac{\partial J}{\partial \hat{x}_j} \cdot (x_j-\mu) \cdot \frac{\partial \sigma^2}{\partial x_i}
\end{aligned}$$

The remaining derivatives are calculated as below,

$$\begin{aligned}
\frac{\partial \mu}{\partial x_i} &= \frac{1}{m} \ \textit{From } (1) \\
\frac{\partial \sigma^2}{\partial x_i} &= \frac{2(x_i-\mu)}{m} \ \textit{From } (2)
\end{aligned}$$

Finally we get,

$$\begin{aligned}
\frac{\partial J}{\partial x_i} &= \frac{\partial J}{\partial \hat{x}_i} \cdot \frac{1}{\sqrt{\sigma^2+\epsilon}} + \frac{1}{m}\sum_{i=1}^{m} \frac{\partial J}{\partial \hat{x}_i} \cdot \frac{-1}{\sqrt{\sigma^2+\epsilon}} - \frac{(\sigma^2+\epsilon)^{-\frac{3}{2}}}{m} \sum_{j=1}^{m} \frac{\partial J}{\partial \hat{x}_j} \cdot (x_j-\mu) \cdot (x_i-\mu) \\
&= \frac{\partial J}{\partial \hat{x}_i} \cdot \frac{1}{\sqrt{\sigma^2+\epsilon}} + \frac{1}{m}\sum_{i=1}^{m} \frac{\partial J}{\partial \hat{x}_i} \cdot \frac{-1}{\sqrt{\sigma^2+\epsilon}} - \frac{1}{m\sqrt{\sigma^2+\epsilon}} \cdot \hat{x}_i \cdot \sum_{j=1}^{m} \frac{\partial J}{\partial \hat{x}_j}\hat{x}_j \\
&= \frac{1}{m\sqrt{\sigma^2+\epsilon}}\left(m\frac{\partial J}{\partial \hat{x}_i} - \sum_{i=1}^{m} \frac{\partial J}{\partial \hat{x}_i} - \hat{x}_i \sum_{j=1}^{m} \frac{\partial J}{\partial \hat{x}_j}\hat{x}_j\right)
\end{aligned}$$

2] Architecture

In Convolutional Neural Networks the bias parameter is associated with each filter and its value is added to the filter output. Such a bias is called a tied bias where the bias remains constant for each location of a feature map, here the learnable parameters is lesser. Untied biases can also be used in which case each location on the input map can have its own bias, the number of learnable parameters drastically increases but the network would now be able to learn location specific information allowing for more fine tuning. Their use depends on the training data, if the data is shifted uniformly over all features a tied bias would work well, but if it has location specific shifts an untied bias can be used for correction.

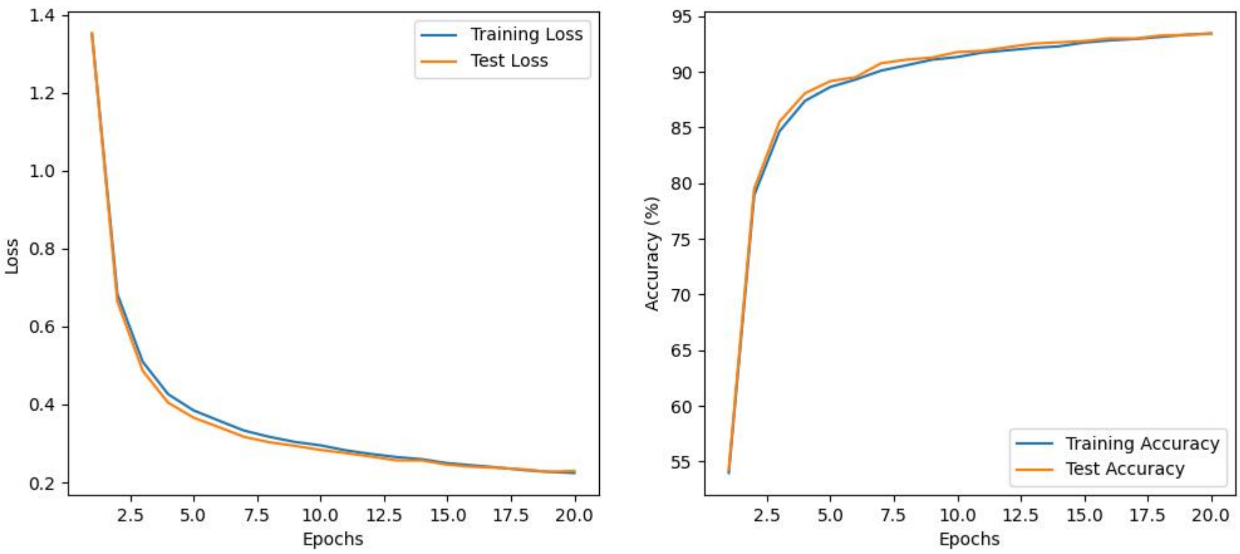
If there is batch normalization operation after the convolution then biasing would be point less as the bias would also get normalized in the case of tied biases. The bias term thus becomes redundant. In case of untied biases the effect will remain as each neuron gets biased differently.

ADA7-ThreeLayer.py

```python
# Implementation of a Three-layer neural network for MNIST classification
# with manual gradient calculation and manual optimization.

import torch
import torch.nn.functional as F
from torch.utils.data import DataLoader
from torchvision import datasets, transforms
import matplotlib.pyplot as plt

torch.manual_seed(0)
lr = 0.005
hidden_dim = 15
batch_size = 64
epochs = 20
plot = True

transform = transforms.Compose([transforms.ToTensor(), transforms.Normalize((0.1307,), (0.3081,))]) # mnist mean and std
train_dataset = datasets.MNIST(root='./data', train=True, download=True, transform=transform)
test_dataset = datasets.MNIST(root='./data', train=False, download=True, transform=transform)
train_loader = DataLoader(dataset=train_dataset, batch_size=batch_size, shuffle=True)
test_loader = DataLoader(dataset=test_dataset, batch_size=1000, shuffle=False)

class ThreeLayerNet:
    def __init__(self, input_size, hidden_size1, hidden_size2, output_size):
        self.W1 = torch.randn(input_size, hidden_size1) * 0.1
        self.b1 = torch.randn(hidden_size1) * 0.1
        self.W2 = torch.randn(hidden_size1, hidden_size2) * 0.1
        self.b2 = torch.randn(hidden_size2) * 0.1
        self.W3 = torch.randn(hidden_size2, output_size) * 0.1
        self.b3 = torch.randn(output_size) * 0.1

    def forward(self, x):
        # Input
        self.x = x
        # Hidden Layer 1
        self.z1 = x @ self.W1 + self.b1
        self.a1 = F.relu(self.z1)
        # Hidden Layer 2
        self.z2 = self.a1 @ self.W2 + self.b2
        self.a2 = F.relu(self.z2)
        # Output Layer
        self.z3 = self.a2 @ self.W3 + self.b3
        return self.z3


model = ThreeLayerNet(784, hidden_dim, hidden_dim, 10)

train_loss_values = []
test_loss_values = []
train_accuracy_values = []
test_accuracy_values = []

def train(epoch):
    for batch_idx, (data, target) in enumerate(train_loader):
```

```
        data = data.view(-1, 784)  # flatten the input

        # === FORWARD PASS ===
        output = model.forward(data)
        log_softmax = F.log_softmax(output, dim=1)
        loss = - torch.mean(log_softmax[range(len(target)), target]) # Equivalent to NLLLoss

        # === BACKWARD PASS ===
        # gradient of the loss w.r.t. output of model
        grad_z3 = F.softmax(output, dim=1)
        grad_z3[range(len(target)), target] -= 1
        grad_z3 /= len(target) # recall that loss is average over batch

        # gradient of the loss w.r.t. the output after the second hidden layer ReLU
        grad_a2 = grad_z3 @ model.W3.T

        # gradient of the loss w.r.t. the output before the second hidden layer ReLU
        grad_z2 = grad_a2.clone()
        grad_z2[model.z2 < 0] = 0

        # gradient of the loss w.r.t. the output after the first hidden layer ReLU
        grad_a1 = grad_z2 @ model.W2.T

        # gradient of the loss w.r.t. the output before the first hidden layer ReLU
        grad_z1 = grad_a1.clone()
        grad_z1[model.z1 < 0] = 0

        # gradient of the loss w.r.t. the model parameters
        model.W3.grad = model.a2.T @ grad_z3
        model.b3.grad = grad_z3.sum(axis=0)
        model.W2.grad = model.a1.T @ grad_z2
        model.b2.grad = grad_z2.sum(axis=0)
        model.W1.grad = model.x.T @ grad_z1
        model.b1.grad = grad_z1.sum(axis=0)

        # === PARAM UPDATES === # Vanilla gradient descent
        model.W3 -= lr * model.W3.grad
        model.b3 -= lr * model.b3.grad
        model.W2 -= lr * model.W2.grad
        model.b2 -= lr * model.b2.grad
        model.W1 -= lr * model.W1.grad
        model.b1 -= lr * model.b1.grad

        # === PRINT EVERY 200 ITERATIONS ===
        if batch_idx % 200 == 0:
            print('Train Epoch: {} [{}/{} ({:.0f}%)]\tLoss: {:.6f}'.format(
                epoch, batch_idx * len(data), len(train_loader.dataset),
                100. * batch_idx / len(train_loader), loss.item()))

@torch.no_grad()
def check(train_or_test, loader):
    loss, correct = 0, 0
    for data, target in loader:
        data = data.view(-1, 784) # flatten the input
        output = model.forward(data)
        log_softmax = F.log_softmax(output, dim=1)
        loss += F.nll_loss(log_softmax, target, reduction='sum').item() # sum up batch loss
        pred = output.argmax(dim=1, keepdim=True) # get the index of the max log-probability
```

```
        correct += pred.eq(target.view_as(pred)).sum().item() # pred is batch_size x 1, target is batch_size

    loss /= len(loader.dataset) # note that reduction is 'sum' instead of 'mean' in F.nll_loss
    accuracy = 100. * correct / len(loader.dataset)
    print('{} set: Average loss: {:.4f}, Accuracy: {}/{} ({:.0f}%)'.format(
        train_or_test, loss, correct, len(loader.dataset), accuracy))
    return loss, accuracy

for epoch in range(1, 1+epochs):
    train(epoch)
    train_loss, train_acc = check(train_or_test='train', loader=train_loader)
    train_loss_values.append(train_loss)
    train_accuracy_values.append(train_acc)
    test_loss, test_acc = check(train_or_test='test', loader=test_loader)
    test_loss_values.append(test_loss)
    test_accuracy_values.append(test_acc)
    print('', end='\n')
    # Notice how we are doing a full forward pass again at the end of each epoch to compute the train in loss and accuracy,
    # but to save time, we could keep track of the loss (or number of correct predictions) for each mini-batch
    # and take an average at the end of the epoch instead.

if plot:
    plt.figure(figsize=(12, 5))
    plt.subplot(1, 2, 1)
    plt.plot(range(1, 1+epochs), train_loss_values, label='Training Loss')
    plt.plot(range(1, 1+epochs), test_loss_values, label='Test Loss')
    plt.xlabel('Epochs')
    plt.ylabel('Loss')
    plt.legend()

    plt.subplot(1, 2, 2)
    plt.plot(range(1, 1+epochs), train_accuracy_values, label='Training Accuracy')
    plt.plot(range(1, 1+epochs), test_accuracy_values, label='Test Accuracy')
    plt.xlabel('Epochs')
    plt.ylabel('Accuracy (%)')
    plt.legend()

    plt.show()
```


Training Loss
Test Loss
Loss
Epochs
Training Accuracy
Test Accuracy
Accuracy (%)
Epochs