Copy_of_Task_02_Optimization_Algorithms

August 28, 2022

1 4. Optimizers

1.1 Introduction to Gradient-descent Optimizers

1.1.1 Model: 1 Hidden Layer Feedforward Neural Network (ReLU Activation)

In this assignment, we are going to train a MLP model (developed using Pytorch) using different Optimization algorithms that have been already discussed in class.

[1]:
```
import torch
import torch.nn as nn
import torchvision.transforms as transforms
import torchvision.datasets as dsets

# Set seed
torch.manual_seed(0)


'''
STEP 1: LOADING DATASET
'''

train_dataset = dsets.MNIST(root='./data', train=True, transform=transforms.
↪ToTensor(), download=True)
test_dataset = dsets.MNIST(root='./data', train=False, transform=transforms.
↪ToTensor())

'''
STEP 2: MAKING DATASET ITERABLE
'''

batch_size = 100
n_iters = 3000
num_epochs = n_iters / (len(train_dataset) / batch_size)
num_epochs = int(num_epochs)

train_loader = torch.utils.data.DataLoader(dataset=train_dataset,␣
↪batch_size=batch_size, shuffle=True)
```

```
test_loader = torch.utils.data.DataLoader(dataset=test_dataset,␣
↪batch_size=batch_size, shuffle=False)

'''
STEP 3: CREATE MODEL CLASS
'''
class FeedforwardNeuralNetModel(nn.Module):
    def __init__(self, input_dim, hidden_dim, output_dim):
        super(FeedforwardNeuralNetModel, self).__init__()
        # Linear function
        self.fc1 = nn.Linear(input_dim, hidden_dim)
        # Non-linearity
        self.relu = nn.ReLU()
        # Linear function (readout)
        self.fc2 = nn.Linear(hidden_dim, output_dim)

    def forward(self, x):
        ### START CODE HERE ###
        # Linear function
        out = self.fc1(x)
        # Non-linearity
        out = self.relu(out)
        # Linear function (readout)
        out = self.fc2(out)
        ### END CODE HERE ###
        return out
'''
STEP 4: INSTANTIATE MODEL CLASS
'''
input_dim = 28*28
hidden_dim = 100
output_dim = 10

model = FeedforwardNeuralNetModel(input_dim, hidden_dim, output_dim)

'''
STEP 5: INSTANTIATE LOSS CLASS
'''
criterion = nn.CrossEntropyLoss()


'''
STEP 6: INSTANTIATE OPTIMIZER CLASS
'''
learning_rate = 0.1

### START CODE HERE ###
```

```python
optimizer = torch.optim.SGD(model.parameters(),lr =learning_rate)
### END CODE HERE ###

'''
STEP 7: TRAIN THE MODEL
'''
iter = 0
for epoch in range(num_epochs):
    for i, (images, labels) in enumerate(train_loader):
        # Load images as Variable
        images = images.view(-1, 28*28).requires_grad_()

        # Clear gradients w.r.t. parameters
        optimizer.zero_grad()

        # Forward pass to get output/logits
        outputs = model(images)

        # Calculate Loss: softmax --> cross entropy loss
        loss = criterion(outputs, labels)

        # Getting gradients w.r.t. parameters
        loss.backward()

        # Updating parameters
        optimizer.step()

        iter += 1

        if iter % 500 == 0:
            # Calculate Accuracy
            correct = 0
            total = 0
            # Iterate through test dataset
            for images, labels in test_loader:
                # Load images to a Torch Variable
                images = images.view(-1, 28*28)

                # Forward pass only to get logits/output
                outputs = model(images)

                # Get predictions from the maximum value
                _, predicted = torch.max(outputs.data, 1)

                # Total number of labels
                total += labels.size(0)
```

```
                # Total correct predictions
                correct += (predicted == labels).sum()

            accuracy = 100 * correct / total

            # Print Loss
            print('Iteration: {}. Loss: {}. Accuracy: {}'.format(iter, loss.
→item(), accuracy))
```

```
Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz to
./data/MNIST/raw/train-images-idx3-ubyte.gz

  0%|          | 0/9912422 [00:00<?, ?it/s]

Extracting ./data/MNIST/raw/train-images-idx3-ubyte.gz to ./data/MNIST/raw

Downloading http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz to
./data/MNIST/raw/train-labels-idx1-ubyte.gz

  0%|          | 0/28881 [00:00<?, ?it/s]

Extracting ./data/MNIST/raw/train-labels-idx1-ubyte.gz to ./data/MNIST/raw

Downloading http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz to
./data/MNIST/raw/t10k-images-idx3-ubyte.gz

  0%|          | 0/1648877 [00:00<?, ?it/s]

Extracting ./data/MNIST/raw/t10k-images-idx3-ubyte.gz to ./data/MNIST/raw

Downloading http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz to
./data/MNIST/raw/t10k-labels-idx1-ubyte.gz

  0%|          | 0/4542 [00:00<?, ?it/s]

Extracting ./data/MNIST/raw/t10k-labels-idx1-ubyte.gz to ./data/MNIST/raw

Iteration: 500. Loss: 0.3460221588611603. Accuracy: 91.5
Iteration: 1000. Loss: 0.21451084315776825. Accuracy: 92.47000122070312
Iteration: 1500. Loss: 0.1919996440410614. Accuracy: 93.87000274658203
Iteration: 2000. Loss: 0.17153751850128174. Accuracy: 94.47000122070312
Iteration: 2500. Loss: 0.11251085251569748. Accuracy: 95.16999816894531
Iteration: 3000. Loss: 0.1736811101436615. Accuracy: 95.5999984741211
```

1.1.2 Optimization Process

`parameters = parameters - learning_rate * parameters_gradients`

1.1.3 Mathematical Interpretation of Gradient Descent

- Model's parameters: $\theta \in^d$
- Loss function: $J(\theta)$
- Gradient w.r.t. parameters: \$ $\nabla J(\theta)\$Learningrate : \eta$
- Batch Gradient descent: $\theta = \theta - \eta \cdot \nabla J(\theta)$

1.2 Optimization Algorithm 1: Batch Gradient Descent

- What we've covered so far: batch gradient descent
  - $\theta = \theta - \eta \cdot \nabla J(\theta)$
- Characteristics
  - Compute the gradient of the lost function w.r.t. parameters for the entire training data, $\nabla J(\theta)$
  - Use this to update our parameters at every iteration
- Problems
  - Unable to fit whole datasets in memory
  - Computationally slow as we attempt to compute a large Jacobian matrix $\rightarrow$ first order derivative, $\nabla J(\theta)$

1.3 Optimization Algorithm 2: Stochastic Gradient Descent

- Modification of batch gradient descent
  - $\theta = \theta - \eta \cdot \nabla J(\theta, x^i, y^i)$
- Characteristics
  - Compute the gradient of the lost function w.r.t. parameters for the **one set of training sample (1 input and 1 label)**, $\nabla J(\theta, x^i, y^i)$
  - Use this to update our parameters at every iteration

1.4 Optimization Algorithm 3: Mini-batch Gradient Descent

- Combination of batch gradient descent & stochastic gradient descent
  - $\theta = \theta - \eta \cdot \nabla J(\theta, x^{i:i+n}, y^{i:i+n})$
- Characteristics
  - Compute the gradient of the lost function w.r.t. parameters for **n sets of training sample (n input and n label)**, $\nabla J(\theta, x^{i:i+n}, y^{i:i+n})$
  - Use this to update our parameters at every iteration
- This is often called SGD in deep learning frameworks

[3]:
```
import torch
import torch.nn as nn
import torchvision.transforms as transforms
import torchvision.datasets as dsets

# Set seed
torch.manual_seed(0)

'''
STEP 1: LOADING DATASET
```

```python
'''

train_dataset = dsets.MNIST(root='./data', train=True, transform=transforms.
 ↪ToTensor(), download=True)
test_dataset = dsets.MNIST(root='./data', train=False, transform=transforms.
 ↪ToTensor())

'''
STEP 2: MAKING DATASET ITERABLE
'''

batch_size = 100
n_iters = 3000
num_epochs = n_iters / (len(train_dataset) / batch_size)
num_epochs = int(num_epochs)

train_loader = torch.utils.data.DataLoader(dataset=train_dataset,␣
 ↪batch_size=batch_size, shuffle=True)
test_loader = torch.utils.data.DataLoader(dataset=test_dataset,␣
 ↪batch_size=batch_size, shuffle=False)

'''
STEP 3: CREATE MODEL CLASS
'''
class FeedforwardNeuralNetModel(nn.Module):
    def __init__(self, input_dim, hidden_dim, output_dim):
        super(FeedforwardNeuralNetModel, self).__init__()
        # Linear function
        self.fc1 = nn.Linear(input_dim, hidden_dim)
        # Non-linearity
        self.relu = nn.ReLU()
        # Linear function (readout)
        self.fc2 = nn.Linear(hidden_dim, output_dim)

    def forward(self, x):
        ### START CODE HERE ###
        # Linear function
        out = self.fc1(x)
        # Non-linearity
        out = self.relu(out)
        # Linear function (readout)
        out = self.fc2(out)
        ### END CODE HERE ###
        return out
'''
STEP 4: INSTANTIATE MODEL CLASS
'''
```

```python
input_dim = 28*28
hidden_dim = 100
output_dim = 10

model = FeedforwardNeuralNetModel(input_dim, hidden_dim, output_dim)

'''
STEP 5: INSTANTIATE LOSS CLASS
'''
criterion = nn.CrossEntropyLoss()


'''
STEP 6: INSTANTIATE OPTIMIZER CLASS
'''
learning_rate = 0.1

### START CODE HERE ###
optimizer = torch.optim.SGD(model.parameters(), lr= learning_rate)
### END CODE HERE ###

'''
STEP 7: TRAIN THE MODEL
'''
iter = 0
for epoch in range(num_epochs):
    for i, (images, labels) in enumerate(train_loader):
        # Load images as Variable
        images = images.view(-1, 28*28).requires_grad_()

        # Clear gradients w.r.t. parameters
        optimizer.zero_grad()

        # Forward pass to get output/logits
        outputs = model(images)

        # Calculate Loss: softmax --> cross entropy loss
        loss = criterion(outputs, labels)

        # Getting gradients w.r.t. parameters
        loss.backward()

        # Updating parameters
        optimizer.step()

        iter += 1
```

```
        if iter % 500 == 0:
            # Calculate Accuracy
            correct = 0
            total = 0
            # Iterate through test dataset
            for images, labels in test_loader:
                # Load images to a Torch Variable
                images = images.view(-1, 28*28).requires_grad_()

                # Forward pass only to get logits/output
                outputs = model(images)

                # Get predictions from the maximum value
                _, predicted = torch.max(outputs.data, 1)

                # Total number of labels
                total += labels.size(0)

                # Total correct predictions
                correct += (predicted == labels).sum()

            accuracy = 100 * correct / total

            # Print Loss
            print('Iteration: {}. Loss: {}. Accuracy: {}'.format(iter, loss.
↪item(), accuracy))
```

```
Iteration: 500. Loss: 0.3460221588611603. Accuracy: 91.5
Iteration: 1000. Loss: 0.21451084315776825. Accuracy: 92.47000122070312
Iteration: 1500. Loss: 0.1919996440410614. Accuracy: 93.87000274658203
Iteration: 2000. Loss: 0.17153751850128174. Accuracy: 94.47000122070312
Iteration: 2500. Loss: 0.11251085251569748. Accuracy: 95.16999816894531
Iteration: 3000. Loss: 0.1736811101436615. Accuracy: 95.5999984741211
```

1.5 Optimization Algorithm 4: SGD Momentum

- Modification of SGD
  - $v_t = \gamma v_{t-1} + \eta \cdot \nabla J(\theta, x^{i:i+n}, y^{i:i+n})$
  - $\theta = \theta - v_t$
- Characteristics
  - Compute the gradient of the lost function w.r.t. parameters for **n sets of training sample (n input and n label)**, $\nabla J(\theta, x^{i:i+n}, y^{i:i+n})$
  - Use this to add to the previous update vector v_{t-1}
  - Momentum, usually set to $\gamma = 0.9$
  - Parameters updated with update vector, v_t that incorporates previous update vector
    - γv_t increases if gradient same sign/direction as v_{t-1}
      - Gives SGD the push when it is going in the right direction (minimizing loss)
      - Accelerated convergence

* γv_t decreases if gradient different sign/direction as v_{t-1}
  - Dampens SGD when it is going in a different direction
  - Lower variation in loss minimization

[4]:
```
import torch
import torch.nn as nn
import torchvision.transforms as transforms
import torchvision.datasets as dsets

# Set seed
torch.manual_seed(0)

'''
STEP 1: LOADING DATASET
'''

train_dataset = dsets.MNIST(root='./data', train=True, transform=transforms.
 ↪ToTensor(), download=True)
test_dataset = dsets.MNIST(root='./data', train=False, transform=transforms.
 ↪ToTensor())

'''
STEP 2: MAKING DATASET ITERABLE
'''

batch_size = 100
n_iters = 3000
num_epochs = n_iters / (len(train_dataset) / batch_size)
num_epochs = int(num_epochs)

train_loader = torch.utils.data.DataLoader(dataset=train_dataset,␣
 ↪batch_size=batch_size, shuffle=True)
test_loader = torch.utils.data.DataLoader(dataset=test_dataset,␣
 ↪batch_size=batch_size, shuffle=False)

'''
STEP 3: CREATE MODEL CLASS
'''
class FeedforwardNeuralNetModel(nn.Module):
    def __init__(self, input_dim, hidden_dim, output_dim):
        super(FeedforwardNeuralNetModel, self).__init__()
        # Linear function
        self.fc1 = nn.Linear(input_dim, hidden_dim)
        # Non-linearity
        self.relu = nn.ReLU()
        # Linear function (readout)
        self.fc2 = nn.Linear(hidden_dim, output_dim)
```

```python
    def forward(self, x):
        ### START CODE HERE ###
        # Linear function
        out = self.fc1(x)
        # Non-linearity
        out = self.relu(out)
        # Linear function (readout)
        out = self.fc2(out)
        ### END CODE HERE ###
        return out
'''
STEP 4: INSTANTIATE MODEL CLASS
'''
input_dim = 28*28
hidden_dim = 100
output_dim = 10

model = FeedforwardNeuralNetModel(input_dim, hidden_dim, output_dim)

'''
STEP 5: INSTANTIATE LOSS CLASS
'''
criterion = nn.CrossEntropyLoss()


'''
STEP 6: INSTANTIATE OPTIMIZER CLASS
'''
learning_rate = 0.1

### START CODE HERE ###

optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate, momentum=0.9)
### END CODE HERE ###

'''
STEP 7: TRAIN THE MODEL
'''
iter = 0
for epoch in range(num_epochs):
    for i, (images, labels) in enumerate(train_loader):
        # Load images as Variable
        images = images.view(-1, 28*28).requires_grad_()

        # Clear gradients w.r.t. parameters
        optimizer.zero_grad()
```

```
        # Forward pass to get output/logits
        outputs = model(images)

        # Calculate Loss: softmax --> cross entropy loss
        loss = criterion(outputs, labels)

        # Getting gradients w.r.t. parameters
        loss.backward()

        # Updating parameters
        optimizer.step()

        iter += 1

        if iter % 500 == 0:
            # Calculate Accuracy
            correct = 0
            total = 0
            # Iterate through test dataset
            for images, labels in test_loader:
                # Load images to a Torch Variable
                images = images.view(-1, 28*28)

                # Forward pass only to get logits/output
                outputs = model(images)

                # Get predictions from the maximum value
                _, predicted = torch.max(outputs.data, 1)

                # Total number of labels
                total += labels.size(0)

                # Total correct predictions
                correct += (predicted == labels).sum()

            accuracy = 100 * correct / total

            # Print Loss
            print('Iteration: {}. Loss: {}. Accuracy: {}'.format(iter, loss.
↪item(), accuracy))
```

```
Iteration: 500. Loss: 0.10879121720790863. Accuracy: 96.01000213623047
Iteration: 1000. Loss: 0.12940317392349243. Accuracy: 96.23999786376953
Iteration: 1500. Loss: 0.1231849417090416. Accuracy: 96.44000244140625
Iteration: 2000. Loss: 0.04057228937745094. Accuracy: 97.52999877929688
Iteration: 2500. Loss: 0.04051990807056427. Accuracy: 97.47000122070312
Iteration: 3000. Loss: 0.18660661578178406. Accuracy: 97.62999725341797
```

1.6 Optimization Algorithm 4: Adam

- Adaptive Learning Rates
  - $m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$
    - Keeping track of decaying gradient
    - Estimate of the mean of gradients
  - $v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$
    - Keeping track of decaying squared gradient
    - Estimate of the variance of gradients
  - When m_t, v_t initializes as 0, $m_t, v_t \to 0$ initially when decay rates small, $\beta_1, \beta_2 \to 1$
    - Need to correct this with:
  - * $ _t = m$_{t\,\overline{1-\beta_1 \$\$ \hat{v}_t = \frac{v_t}{1-\beta_2} \$}}$
  - * $\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t + \epsilon}} \hat{m}_t$
  - Default recommended values
    - $\beta_1 = 0.9$
    - $\beta_2 = 0.999$
    - $\epsilon = 10^{-8}$
- Instead of learning rate $\to$ equations account for estimates of mean/variance of gradients to determine the next learning rate

[7]:
```python
import torch
import torch.nn as nn
import torchvision.transforms as transforms
import torchvision.datasets as dsets

# Set seed
torch.manual_seed(0)

'''
STEP 1: LOADING DATASET
'''

train_dataset = dsets.MNIST(root='./data', train=True, transform=transforms.
↪ToTensor(), download=True)
test_dataset = dsets.MNIST(root='./data', train=False, transform=transforms.
↪ToTensor())

'''
STEP 2: MAKING DATASET ITERABLE
'''

batch_size = 100
n_iters = 3000
num_epochs = n_iters / (len(train_dataset) / batch_size)
num_epochs = int(num_epochs)
```

```
train_loader = torch.utils.data.DataLoader(dataset=train_dataset,␣
↪batch_size=batch_size, shuffle=True)
test_loader = torch.utils.data.DataLoader(dataset=test_dataset,␣
↪batch_size=batch_size, shuffle=False)

'''
STEP 3: CREATE MODEL CLASS
'''
class FeedforwardNeuralNetModel(nn.Module):
    def __init__(self, input_dim, hidden_dim, output_dim):
        super(FeedforwardNeuralNetModel, self).__init__()
        # Linear function
        self.fc1 = nn.Linear(input_dim, hidden_dim)
        # Non-linearity
        self.relu = nn.ReLU()
        # Linear function (readout)
        self.fc2 = nn.Linear(hidden_dim, output_dim)

    def forward(self, x):
        ### START CODE HERE ###
        # Linear function
        out = self.fc1(x)
        # Non-linearity
        out = self.relu(out)
        # Linear function (readout)
        out = self.fc2(out)
        ### END CODE HERE ###
        return out
'''
STEP 4: INSTANTIATE MODEL CLASS
'''
input_dim = 28*28
hidden_dim = 100
output_dim = 10

model = FeedforwardNeuralNetModel(input_dim, hidden_dim, output_dim)

'''
STEP 5: INSTANTIATE LOSS CLASS
'''
criterion = nn.CrossEntropyLoss()


'''
STEP 6: INSTANTIATE OPTIMIZER CLASS
'''
# learning_rate = 0.001
```

```python
### START CODE HERE ###
optimizer = torch.optim.Adam(model.parameters(),lr=0.001, betas=(0.9, 0.999),␣
↪eps=1e-08)
### END CODE HERE ###

'''
STEP 7: TRAIN THE MODEL
'''
iter = 0
for epoch in range(num_epochs):
    for i, (images, labels) in enumerate(train_loader):
        # Load images as Variable
        images = images.view(-1, 28*28).requires_grad_()

        # Clear gradients w.r.t. parameters
        optimizer.zero_grad()

        # Forward pass to get output/logits
        outputs = model(images)

        # Calculate Loss: softmax --> cross entropy loss
        loss = criterion(outputs, labels)

        # Getting gradients w.r.t. parameters
        loss.backward()

        # Updating parameters
        optimizer.step()

        iter += 1

        if iter % 500 == 0:
            # Calculate Accuracy
            correct = 0
            total = 0
            # Iterate through test dataset
            for images, labels in test_loader:
                # Load images to a Torch Variable
                images = images.view(-1, 28*28)

                # Forward pass only to get logits/output
                outputs = model(images)

                # Get predictions from the maximum value
                _, predicted = torch.max(outputs.data, 1)
```

```
                # Total number of labels
                total += labels.size(0)

                # Total correct predictions
                correct += (predicted == labels).sum()

            accuracy = 100 * correct / total

            # Print Loss
            print('Iteration: {}. Loss: {}. Accuracy: {}'.format(iter, loss.
↪item(), accuracy))
```

```
Iteration: 500. Loss: 0.2257964313030243. Accuracy: 93.30999755859375
Iteration: 1000. Loss: 0.17263264954090118. Accuracy: 94.72000122070312
Iteration: 1500. Loss: 0.136827290058136. Accuracy: 95.54000091552734
Iteration: 2000. Loss: 0.07791975140571594. Accuracy: 96.41000366210938
Iteration: 2500. Loss: 0.07298330217599869. Accuracy: 96.9000015258789
Iteration: 3000. Loss: 0.1346699446439743. Accuracy: 97.20999908447266
```

1.7 Other Adaptive Algorithms

- Other adaptive algorithms (like Adam, adapting learning rates)
  - Adagrad
  - Adadelta
  - Adamax
  - RMSProp