1 Part I

1.1 Introduction

Gradient descent is one of the most popular and widely used optimization algorithms in machine learning. It serves as the backbone for many other optimization algorithms, such as *stochastic gradient descent (SGD)*. This project aims to study *gradient descent* and its variants to understand their role in optimization algorithms. The problem we would like to optimize can be formulated as:

$$\min_{x \in \mathbb{R}^d} f(x) + R(x),$$

where:

- $f(x)$ is a convex, smooth, and differentiable function, representing the cost function (the primary objective).
- $R(x)$ is a proximable regularization term, which may not necessarily be differentiable.

So given the problem, the updates to the model weights (x) can be formulated using *gradient descent* as:

$$x_{t+1} = x_t - \eta \nabla f(x_t)$$

where:

- x_t : Current parameters
- η : Learning rate
- $\nabla f(x_t)$: Gradient of the loss function with respect to parameters

1.1.1 Stochastic Gradient Descent

In many real-world scenarios, machine learning models must work with large datasets, making full-batch computations computationally inconceivable. *Stochastic gradient descent (SGD)* plays a crucial role in addressing this challenge. Instead of processing the entire dataset at once, *SGD* uses a randomly selected subset of data points, significantly reducing the computational burden. The objective function in such cases can be represented as:

$$f(x) = \mathbb{E}_{\xi \sim \mathcal{D}} \left[f_\xi(x) \right],$$

where:

- ξ is a random variable sampled from the data distribution $\mathcal{D}$,
- $f_\xi : \mathbb{R}^d \to \mathbb{R}$ is a smooth function for each ξ.
- $\mathbb{E}_{\xi \sim \mathcal{D}} \left[f_\xi(x) \right]$: We expect the objective function calculated using a subset of the distribution same as the one calculated using the entire dataset.

Several variants of stochastic gradient descent have been proposed to address specific challenges. For instance:

- Proximal SGD is designed to handle non-differentiable regularization terms.

- Minibatch SGD uses batches of data and trains the model.
- SVRG (Stochastic Variance Reduced Gradient) aims to reduce the variance of gradient estimates.

In this project, I have incorporated the L1 penalty (which is non-differentiable) and implemented proximal SGD alongside basic SGD. A detailed discussion of these algorithms will follow in later sections.

1.1.2 Role of Stochastic Gradient Descent in Distributed Training

The idea of training from a subset of data points is extended to distribute training across multiple devices. *Federated learning* is one such class of algorithms that leverages this concept. In *federated learning*, a model is trained collaboratively across several machines, where the data is partitioned among these devices.

A *master* device maintains the global model, while *client* devices receive the global model, train it locally using their own data, and then send the computed updates back to the *master* device. The *master* aggregates these updates to improve the global model iteratively.

The parameter update rule in such a distributed system can be expressed as:

$$x_{t+1} = x_t - \eta \frac{1}{N} \sum_{i=1}^{N} \nabla f_i(x_t)$$

where:

- N : Number of devices
- $f_i(x_t)$: Loss function for data on device i

1.1.3 Project Prerequisites

To build a strong foundation for the concepts covered in this project, I highly recommend exploring the playlist Optimization Algorithms by Professor Constantine Caramanis. This playlist offers an excellent introduction to the foundational ideas implemented in this project and includes proofs of convergence, which help explain why specific algorithms work and why they exist.

Additionally, the playlist provides an overview of *Linear Algebra* and the necessary concepts required to develop an intuitive understanding of the optimization methods discussed in this project.

The following resources have also been used to understand algorithms discussed in this project:

- A Unified Theory of SGD: Variance Reduction, Sampling, Quantization and Coordinate Descent
- Proximal Algorithms
- Proximal SGD with Linear Search
- Convex Optimization
- Proximal Gradient Descent

For the implementation, I utilized the *MNIST* dataset, a standard dataset widely used in machine learning for image classification tasks. The algorithms were implemented in PyTorch, where I wrote custom optimizers and used them to train the model. To run these algorithms efficiently, I made use of a cloud-based RTX 3060 GPU.

```python
import torch
from torch import nn
from torchvision import datasets
import fastai
from torchvision.transforms import ToTensor
# from fastai.data.core import DataLoader
from torch.utils.data import DataLoader
# from fastai.data.core import DataLoaders
# from fastai.callback.core import Callback
# from fastai.vision.all import Learner, Metric
# from fastai import optimizer
import torch.nn.functional as F
```

```python
training_data = datasets.MNIST(
    root="data",
    train=True,
    download=True,
    transform=ToTensor(),
)

# Download test data from open datasets.
test_data = datasets.MNIST(
    root="data",
    train=False,
    download=True,
    transform=ToTensor(),
)
```

```python
batch_size = 256

# Create data loaders.
train_dataloader = DataLoader(training_data, batch_size=batch_size)
test_dataloader = DataLoader(test_data, batch_size=batch_size)

for X, y in test_dataloader:
    print(f"Shape of X [N, C, H, W]: {X.shape}")
    print(f"Shape of y: {y.shape} {y.dtype}")
    break
```

```
Shape of X [N, C, H, W]: torch.Size([256, 1, 28, 28])
Shape of y: torch.Size([256]) torch.int64
```

```python
# Get cpu, gpu or mps device for training.
device = (
    "cuda"
    if torch.cuda.is_available()
    else "mps"
    if torch.backends.mps.is_available()
    else "cpu"
)
print(f"Using {device} device")
```

```
Using cuda device
```

```python
class NeuralNetwork(nn.Module):
    def __init__(self):
        super().__init__()
        self.flatten = nn.Flatten()
        self.linear_relu_stack = nn.Sequential(
            nn.Linear(28*28, 512),
            nn.ReLU(),
            nn.Linear(512, 512),
            nn.ReLU(),
            nn.Linear(512, 10)
        )

    def forward(self, x):
        x = self.flatten(x)
        logits = self.linear_relu_stack(x)
        return logits

model = NeuralNetwork().to(device)
model
```

```
NeuralNetwork(
  (flatten): Flatten(start_dim=1, end_dim=-1)
  (linear_relu_stack): Sequential(
    (0): Linear(in_features=784, out_features=512, bias=True)
    (1): ReLU()
    (2): Linear(in_features=512, out_features=512, bias=True)
    (3): ReLU()
    (4): Linear(in_features=512, out_features=10, bias=True)
  )
)
```

```python
# CrossEntropyLoss for Classification tasks
loss_fn = nn.CrossEntropyLoss()
```

1.1.4 Basic Stochastic Gradient Descent

Every custom optimization class requires a constructor that initializes the class with the model parameters and the hyperparameters necessary for the optimization process. In this implementation:

- *self.params* represents the model parameters that are updated in each iteration.
- *self.hypers* stores the hyperparameters, such as the learning rate (η), used during optimization.

A step function must also be implemented. In the context of Stochastic Gradient Descent (SGD), this function updates the model parameters using the following rule:

$$x_{t+1} = x_t - \eta \nabla f(x_t),$$

where:

- x_t is the parameter at iteration t ,
- η is the learning rate,
- $\nabla f(x_t)$ is the gradient of the loss function with respect to the parameter x_t .

In the case of SGD, this update is performed on a randomly selected subset (batch) of data, rather than the entire dataset, to reduce computational cost.

The zero_grad function is implemented to reset the gradients of the parameters to None before calculating new gradients. This step is essential because PyTorch accumulates gradients by default during backpropagation. Resetting gradients ensures that each optimization step is based on the current computation only.

```python
class SGDBasic:
    def __init__(self, params, lr):
        self.params,self.lr = list(params),lr
        self.state = {p: {} for p in self.params}
        self.hypers = [{'lr': lr}]

    def step(self, *args, **kwargs):
        for p in self.params:
            p.data -= p.grad.data * self.lr

    def zero_grad(self, *args, **kwargs):
        for p in self.params:
            p.grad = None

    def set_hypers(self, **kwargs):
        if 'lr' in kwargs:
            self.lr = kwargs['lr']
            self.hypers[0]['lr'] = kwargs['lr']

# optimizer = BasicOptimizer(model.parameters(), lr=1e-3)
```

1.1.5 Proximal Gradient Descent

Proximal Gradient Descent extends the basic gradient descent algorithm by incorporating a regularization term R using its proximal operator. Instead of directly calculating and updating the parameters, the algorithm applies the proximal operator of R to the new iterate. The update step is expressed as:

$$x_{t+1} = \text{prox}_{\gamma R}(x_t - \gamma g_t),$$

where:

- g_t is an unbiased estimator of the gradient (i.e., a stochastic gradient),

$$\mathbb{E}[g_t|x_t] = \nabla f(x_t),$$

ensuring that g_t approximates the true gradient $\nabla f(x_t)$ in expectation.

- The proximal operator $\text{prox}_{\gamma R}(x)$ is defined as:

$$\text{prox}_{\gamma R}(x) := \arg\min_u \left\{ \gamma R(u) + \frac{1}{2}|u - x|^2 \right\}.$$

This method is particularly useful when R is non-differentiable (e.g., $L1 - regularization$), as it allows for the efficient handling of such terms while ensuring convergence properties.

In this project, I make use of the L1 regularizer, which promotes sparsity in the model weights. The proximal operator for L1 regularization simplifies to the soft-thresholding operator (refer Proximal algorithms page 177), defined as:

$$\text{soft\_threshold}(x, \lambda) = \begin{cases} x - \lambda, & x > \lambda \\ 0, & |x| \leq \lambda \\ x + \lambda, & x < -\lambda \end{cases}$$

This encourages small parameter values to shrink to zero, effectively reducing the model complexity and which could be beneficial in reducing bandwidth while communicating model weights.

For this variant of *Proximal SGD*, I make use of a constant learning rate (γ). While this approach simplifies the implementation and training process, it cannot guarantee a reduction in the objective function value across iterations, as convergence often requires an adaptive or decaying learning rate. Despite this limitation, the method is effective for exploring the impact of proximal operators on optimization with sparsity constraints.

```python
# lr corresponds to gamma
class ProxSGD:
    def __init__(self, params, lr):
        self.params, self.lr = list(params), lr
        self.state = {p: {} for p in self.params}
        self.hypers = [{'lr': lr}]
    def soft_threshold(self, x, eta):
        # Apply the soft-thresholding operator
        return F.softshrink(x, lambd=eta)

    def prox_operator(self, x, eta):
        # Use the soft-thresholding operator as the proximal step
        return self.soft_threshold(x, eta)

    def Gt(self, x, eta, x_grad):
        return (1/self.lr) * (x - self.prox_operator(x - self.lr * x_grad, eta))

    def step(self, *args, **kwargs):
        for p in self.params:
            if p.grad is not None:  # Ensure gradients exist
                p.data -= self.lr * self.Gt(p.data, 1e-5, p.grad.data)

    def zero_grad(self, *args, **kwargs):
        for p in self.params:
            p.grad = None

    def set_hypers(self, **kwargs):
        if 'lr' in kwargs:
            self.lr = kwargs['lr']
            self.hypers[0]['lr'] = kwargs['lr']
```

1.1.6 Proximal Gradient Descent with Line Search

In the standard Proximal Gradient Descent algorithm (as implemented previously), the learning rate (γ) is constant. While this simplifies the algorithm, it can lead to suboptimal performance when the constant step size fails to guarantee a reduction in the objective function. This limitation is particularly problematic when the objective function is non-smooth or exhibits steep gradients.

The Proximal Gradient Descent with Line Search algorithm addresses this issue by dynamically adjusting the step size (γ) during training. This ensures that each update reduces the loss function.

Refer the following resources to get a more clear understanding for this algorithm:

- Proximal Algorithms Page 148
- Proximal SGD with Linear Search : An alternative implementation of the algorithm
- Proximal Gradient Descent : Youtube video for the explanation of the algorithm

There are different ways to determine a step size (γ) that reduces the cost function. In this implementation, I use a basic approach by stopping the algorithm at a value where I get reduced cost, but a more sophisticated version is described by *Beck and Teboulle* in their work on Proximal Algorithms. The method ensures that the step size is adaptively adjusted to guarantee convergence.

```python
class ProxSGDWithLinearSearch:
    def __init__(self, params, lr):
        self.params, self.lr = list(params), lr
        self.state = {p: {} for p in self.params}
        self.hypers = [{'lr': lr}]
        self.max_iter = 5
        self.eta = 1e-5

    def soft_threshold(self, x, eta):
        # Apply the soft-thresholding operator
        return F.softshrink(x, lambd=eta)

    def prox_operator(self, x):
        # Use the soft-thresholding operator as the proximal step
        return self.soft_threshold(x, self.eta)

    def Gt(self, x, step_size, x_grad):
        return (1/step_size) * (x - self.prox_operator(x - step_size * x_grad))

    def step(self, *args, **kwargs):
        model = kwargs.get("model")
        loss_fn = kwargs.get("loss_fn")
        X = kwargs.get("X")
        y = kwargs.get("y")

        orig_params = [p.data.clone() for p in self.params]
        step_size = self.lr
        with torch.no_grad():
            pred = model(X)
            old_loss = loss_fn(pred, y)

        for _ in range(self.max_iter):
            for p in self.params:
                if p.grad is not None:
                    Gt_val = self.Gt(p.data, step_size, p.grad.data)
                    p.data = p.data - step_size * Gt_val
            with torch.no_grad():
                pred = model(X)
                new_loss = loss_fn(pred, y)
            if new_loss < old_loss:
                break
            else:
                for i, j in zip(self.params, orig_params):
                    i.data.copy_(j)
                step_size *= 0.5
        else:
            for p in self.params:
                if p.grad is not None:
                    Gt_val = self.Gt(p.data, step_size, p.grad.data)
                    p.data = p.data - step_size * Gt_val
        self.lr = step_size
    def zero_grad(self, *args, **kwargs):
        for p in self.params:
            p.grad = None

    def set_hypers(self, **kwargs):
        if 'lr' in kwargs:
            self.lr = kwargs['lr']
            self.hypers[0]['lr'] = kwargs['lr']
```

```
def train(dataloader, model, loss_fn, optimizer):
    size = len(dataloader.dataset)
    model.train()
    for batch, (X, y) in enumerate(dataloader):
        X, y = X.to(device), y.to(device)

        # Compute prediction error
        pred = model(X)
        loss = loss_fn(pred, y)

        # Backpropagation
        loss.backward()
        optimizer.step(model=model, loss_fn=loss_fn, X=X, y=y)
        optimizer.zero_grad()

        if batch % 100 == 0:
            print(f"BATCH: {batch} of {size/batch_size} batches")
            loss, current = loss.item(), (batch + 1) * len(X)
            print(f"loss: {loss:>7f}  [{current:>5d}/{size:>5d}]")
```

```
def test(dataloader, model, loss_fn):
    size = len(dataloader.dataset)
    num_batches = len(dataloader)
    model.eval()
    test_loss, correct = 0, 0
    with torch.no_grad():
        for X, y in dataloader:
            X, y = X.to(device), y.to(device)
            pred = model(X)
            test_loss += loss_fn(pred, y).item()
            correct += (pred.argmax(1) == y).type(torch.float).sum().item()
    test_loss /= num_batches
    correct /= size
    print(f"Test Error: \n Accuracy: {(100*correct):>0.1f}%, Avg loss:␣
↪{test_loss:>8f} \n")
```

```
for i in range(5):
    train(train_dataloader, model, loss_fn, ProxSGDWithLinearSearch(model.
↪parameters(), 20))
```

```
BATCH: 0 of 234.375 batches
loss: 2.291508  [  256/60000]
BATCH: 100 of 234.375 batches
loss: 0.352378  [25856/60000]
BATCH: 200 of 234.375 batches
loss: 0.315699  [51456/60000]
BATCH: 0 of 234.375 batches
loss: 0.289511  [  256/60000]
BATCH: 100 of 234.375 batches
loss: 0.245460  [25856/60000]
BATCH: 200 of 234.375 batches
loss: 0.208257  [51456/60000]
BATCH: 0 of 234.375 batches
loss: 0.204918  [  256/60000]
BATCH: 100 of 234.375 batches
loss: 0.208769  [25856/60000]
BATCH: 200 of 234.375 batches
loss: 0.162873  [51456/60000]
BATCH: 0 of 234.375 batches
loss: 0.168498  [  256/60000]
BATCH: 100 of 234.375 batches
loss: 0.185506  [25856/60000]
BATCH: 200 of 234.375 batches
loss: 0.139565  [51456/60000]
BATCH: 0 of 234.375 batches
loss: 0.143701  [  256/60000]
BATCH: 100 of 234.375 batches
loss: 0.167683  [25856/60000]
BATCH: 200 of 234.375 batches
loss: 0.125072  [51456/60000]
```

```
test(test_dataloader, model, loss_fn)
```

```
Test Error:
 Accuracy: 95.4%, Avg loss: 0.149815
```