Constrained Gradient Descent And Variants

November 19, 2024

1 Different Perspective of Gradient Descent: Constrained Optimization & Variants of Gradient Descent Re-derived from Taylor Theory

[37]:
```
import numpy as np
from tensorflow.keras.datasets import mnist
import matplotlib.pyplot as plt
```

Goal is to differentiate number 4's and number 9's in the MNIST data set

[38]:
```
def load_mnist_4_and_9():
    '''Load MNIST data with only 4 and 9 digits'''
    (x_train, y_train), (x_test, y_test) = mnist.load_data()

    # Filter for 4 and 9 digits
    mask_train = np.isin(y_train, [4, 9])
    mask_test = np.isin(y_test, [4, 9])
    x_train, y_train = x_train[mask_train], y_train[mask_train]
    x_test, y_test = x_test[mask_test], y_test[mask_test]

    # Flatten and normalize data
    x_train, x_test = x_train.reshape((-1, 784)) / 255.0, x_test.reshape((-1,␣
↪784)) / 255.0
    y_train = np.where(y_train == 4, 0, 1)
    y_test = np.where(y_test == 4, 0, 1)
    return x_train, y_train, x_test, y_test
```

2 Objective Functions

The objective function (F(w)) for logistic regression is:

$$F(w) = \frac{1}{N}\sum_{i=1}^{N}\log(1+e^{-y_i\langle w,x_i\rangle})$$

where: - $ N $ is the number of training samples, - $ y_i $ is the label for sample $ i $ (1 or -1), - $ x_i $ is the feature vector for sample $ i $, - $ w $ is the weight vector.

[39]:
```python
def binary_cross_entropy_loss(w, X, y):
    '''Binary cross entropy loss and gradient'''
    logits = X @ w
    predictions = 1 / (1 + np.exp(-logits))
    loss = -np.mean(y * np.log(predictions + 1e-8) + (1 - y) * np.log(1 -␣
 ↪predictions + 1e-8))
    grad = X.T @ (predictions - y) / len(y)
    return loss, grad

def logistic_loss(w, X, y):
    margins = y * (X @ w)
    loss = np.mean(np.log(1 + np.exp(-margins)))
    grad = -np.mean((y / (1 + np.exp(margins)))[:, np.newaxis] * X, axis=0)
    return loss, grad
```

3 L-2 Gradient Descent

The update rule for pure gradient descent is:

$$w(t+1) = w(t) - \mu \nabla F(w(t))$$

[40]:
```python
def gradient_descent(X, y, step_size=1e-4, num_iterations=2000):
    '''Vanilla gradient descent with logistic loss'''
    w = np.zeros(X.shape[1])
    losses = []
    for t in range(num_iterations):
        loss, grad = logistic_loss(w, X, y)
        w -= step_size * grad
        losses.append(loss)
    return w, losses
```

4 L-2 BackTracking LineSearch Gradient Descent

The Armijo condition for gradient descent is:

$$F(w(t) - \mu \nabla F(w(t))) \leq F(w(t)) - c \cdot \mu \|\nabla F(w(t))\|^2$$

where \$ c \$ is a small constant, \$ \$ is the step size, and \$ | F(w(t))| \$ represents the norm of the gradient at \$ w(t) \$.

[41]:
```python
def gradient_descent_armijo(X, y, initial_step_size=1e-2, num_iterations=2000,␣
 ↪beta=0.8, c=0.1, max_while_iters=10):
    """Armijo condition constrained Backtracking LineSearch gradient descent␣
 ↪with logistic loss."""
    w = np.zeros(X.shape[1])
    losses = []

    for t in range(num_iterations):
        loss, grad = binary_cross_entropy_loss(w, X, y)
        step_size = initial_step_size
        while_iters = 0  # Track number of while-loop iterations

        # Armijo condition line search
        while while_iters < max_while_iters:
            new_w = w - step_size * grad
            new_loss, _ = binary_cross_entropy_loss(new_w, X, y)
            if new_loss <= loss - c * step_size * np.sum(grad**2):
                break
            step_size *= beta
            while_iters += 1

        # Check if Armijo condition was not met after maximum attempts
        if while_iters == max_while_iters:
            print("Warning: Armijo condition not satisfied within maximum␣
↪iterations.")

        if t % 100 == 0:
            print(f'Learning rate at {t} is {step_size}')
        # Update weights
        w = new_w
        losses.append(loss)

    return w, losses
```

5 L-Inf Gradient Descent

L-inf Gradient descent is defiend as the following and uses this "gradient":

Notice that these are analytical answer derived from the constrained GD problem

$$w(t+1) = w(t) - \mu p(t)$$

Where

$$p(t) = \text{sign}(\nabla F(w(t))) \cdot \|\nabla F(w(t))\|_1$$

Duo with the L1 norm problem: 1. We uniformally step towards all directions in the vector

[42]:
```
def l_inf_gradient_descent(X, y, step_size=1e-6, num_iterations=2000):
    ''' L∞ Gradient Descent'''

    w = np.zeros(X.shape[1])
    losses = []
    for t in range(num_iterations):
        loss, grad = logistic_loss(w, X, y)
        p = np.sign(grad) * np.linalg.norm(grad, ord=1)
        w -= step_size * p
        losses.append(loss)
    return w, losses
```

6 L-1 Gradient Descent

L-one Gradient descent is defiend as the following and uses steps:

$$w(t+1) = w(t) - \mu p(t)$$

Where

$$p(t) = \text{sign}(\nabla_{j^*} F(w(t))) \cdot \|\nabla F(w(t))\|_\infty e_{j^*}$$

Respect the axis, jig-saw coordinate descent, instead of looking at a sphere, we look at a trainagle 1. Step direction is sparse 2. At each step we only step towards one direction in the vector

[43]:
```
def l1_gradient_descent(X, y, step_size=1e-4, num_iterations=2000):
    '''L1 Gradient Descent (Coordinate Descent)'''

    w = np.zeros(X.shape[1])
    losses = []
    frequency_counter = np.zeros(X.shape[1])

    for t in range(num_iterations):
        loss, grad = logistic_loss(w, X, y)
        j_star = np.argmax(np.abs(grad))  # index of max gradient entry
        p = np.zeros(X.shape[1])
        p[j_star] = np.sign(grad[j_star]) * np.linalg.norm(grad, ord=np.inf) #␣
 ↪L1 norm, step in only one direction
        w -= step_size * p
        losses.append(loss)

        # Track which max grad index that was selected
        frequency_counter[j_star] += 1

    return w, losses, frequency_counter
```

7 Gradeint Descent With Momentum

The momentum-based update rule is given by:

$$x^{(t+1)} = x^{(t)} - \mu \nabla f(x^{(t)}) + \beta \left(x^{(t)} - x^{(t-1)}\right)$$

Where: - $x^{(t)}$ represents the parameter vector at iteration t, - μ is the learning rate, - $\nabla f(x^{(t)})$ is the gradient of the loss function with respect to $\chi^{(t)}$, - β is the momentum coefficient, - $x^{(t-1)}$ is the parameter vector from the previous iteration.

[44]:
```
def momentum_gradient_descent(X, y, step_size=1e-5, momentum=0.9,␣
↪num_iterations=2000):
    """
    Gradient Descent with Momentum
    """
    w = np.zeros(X.shape[1])
    velocity = np.zeros(X.shape[1]) # first iter velocity is 0, start getting␣
↪after second iter
    losses = []

    for _ in range(num_iterations):
        loss, grad = logistic_loss(w, X, y)
        velocity = momentum * velocity + step_size * grad
        w -= velocity
        losses.append(loss)

    return w, losses
```

8 Gradient Descent With Nestrov Acceleration

The Nesterov Accelerated Gradient Descent (NAG) algorithm updates weights in two steps:

1. **Lookahead step**: Following the velocity to walk a bit more

$$y^{(t+1)} = x^{(t)} + \beta \left(x^{(t)} - x^{(t-1)}\right)$$

   Here, $y^{(t+1)}$ is the "lookahead" position, where:

2. **Update step**:

$$x^{(t+1)} = y^{(t+1)} - \mu \nabla f(y^{(t+1)})$$

[53]:
```
def nesterov_accelerated_gradient_descent(X, y, step_size=1e-5, momentum=0.9,␣
↪num_iterations=2000):
    """
    Gradient Descent with  Nesterov Acceleration
    """
    x = np.zeros(X.shape[1])
    velocity = np.zeros(X.shape[1])
    losses = []

    for _ in range(num_iterations):
        # Look ahead using momentum
        lookahead_y = x - momentum * velocity

        # Compute loss and gradient at the look-ahead position
        loss, grad = logistic_loss(lookahead_y, X, y)
        velocity = momentum * velocity + step_size * grad
        w -= velocity
        losses.append(loss)

    return w, losses
```

9 Execution

[46]:
```
x_train, y_train, x_test, y_test = load_mnist_4_and_9()

_, l_inf_losses = l_inf_gradient_descent(x_train, y_train)
_, l1_losses, frequency_counter = l1_gradient_descent(x_train, y_train)
_, l2_losses = gradient_descent(x_train, y_train)
```

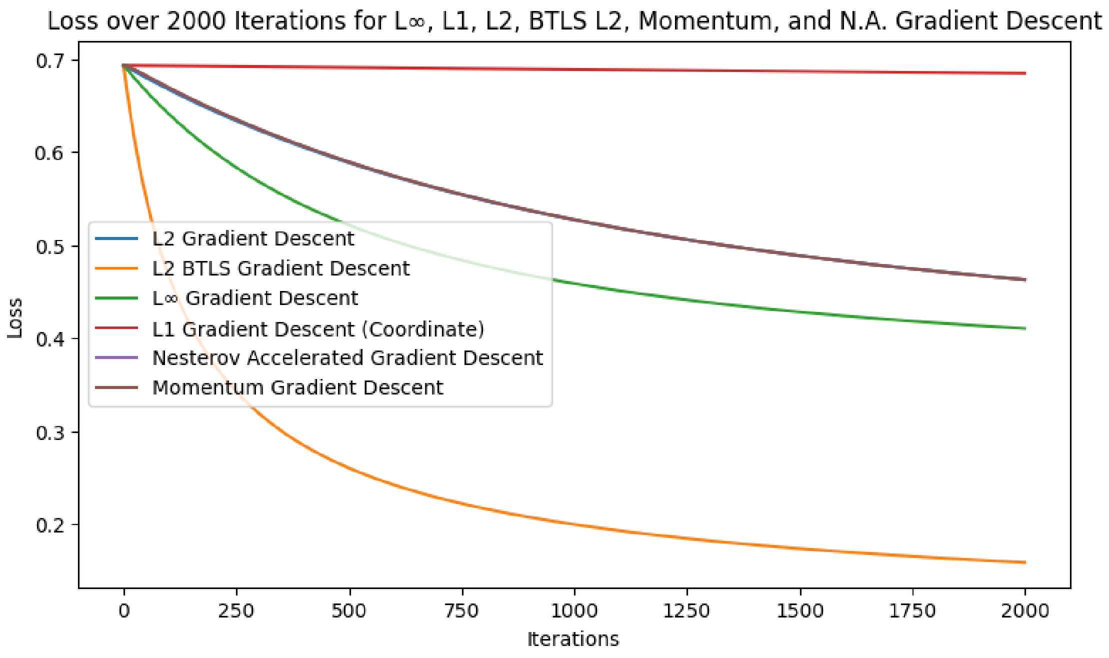
[47]:
```
_, na_losses = nesterov_accelerated_gradient_descent(x_train, y_train)
_, m_losses = momentum_gradient_descent(x_train, y_train)
```

[48]:
```
_, btls_losses = gradient_descent_armijo(x_train, y_train)
```

```
Learning rate at 0 is 0.01
Learning rate at 100 is 0.01
Learning rate at 200 is 0.01
Learning rate at 300 is 0.01
Learning rate at 400 is 0.01
Learning rate at 500 is 0.01
Learning rate at 600 is 0.01
Learning rate at 700 is 0.01
Learning rate at 800 is 0.01
Learning rate at 900 is 0.01
Learning rate at 1000 is 0.01
Learning rate at 1100 is 0.01
Learning rate at 1200 is 0.01
Learning rate at 1300 is 0.01
Learning rate at 1400 is 0.01
Learning rate at 1500 is 0.01
Learning rate at 1600 is 0.01
Learning rate at 1700 is 0.01
Learning rate at 1800 is 0.01
Learning rate at 1900 is 0.01
```

10 Plot the loss over iterations

[49]:
```
plt.figure(figsize=(9, 5))
plt.plot(l2_losses, label="L2 Gradient Descent")
plt.plot(btls_losses, label="L2 BTLS Gradient Descent")
plt.plot(l_inf_losses, label="L∞ Gradient Descent")
plt.plot(l1_losses, label="L1 Gradient Descent (Coordinate)")
plt.plot(na_losses, label="Nesterov Accelerated Gradient Descent")
plt.plot(m_losses, label="Momentum Gradient Descent")
plt.xlabel("Iterations")
plt.ylabel("Loss")
plt.title("Loss over 2000 Iterations for L∞, L1, L2, BTLS L2, Momentum, and N.A.
  ↪ Gradient Descent")
plt.legend()
plt.show()
```

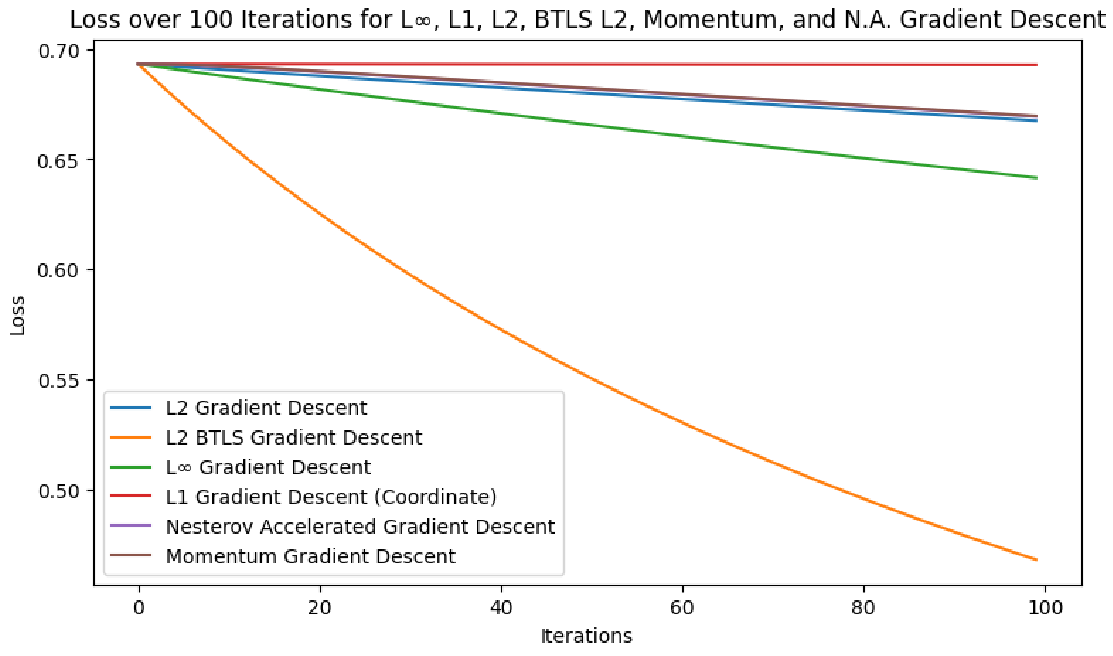


[50]:
```
x_train, y_train, x_test, y_test = load_mnist_4_and_9()
num_iterations = 100
_, l_inf_losses_100 = l_inf_gradient_descent(x_train, y_train,␣
  ↪num_iterations=num_iterations)
_, l1_losses_100, frequency_counter = l1_gradient_descent(x_train, y_train,␣
  ↪num_iterations=num_iterations)
_, l2_losses_100 = gradient_descent(x_train, y_train,␣
  ↪num_iterations=num_iterations)
```

```
_, na_losses_100 = nesterov_accelerated_gradient_descent(x_train, y_train,␣
↪num_iterations=num_iterations)
_, m_losses_100 = momentum_gradient_descent(x_train, y_train,␣
↪num_iterations=num_iterations)
_, btls_losses_100 = gradient_descent_armijo(x_train, y_train,␣
↪num_iterations=num_iterations)
```

```
Learning rate at 0 is 0.01
```

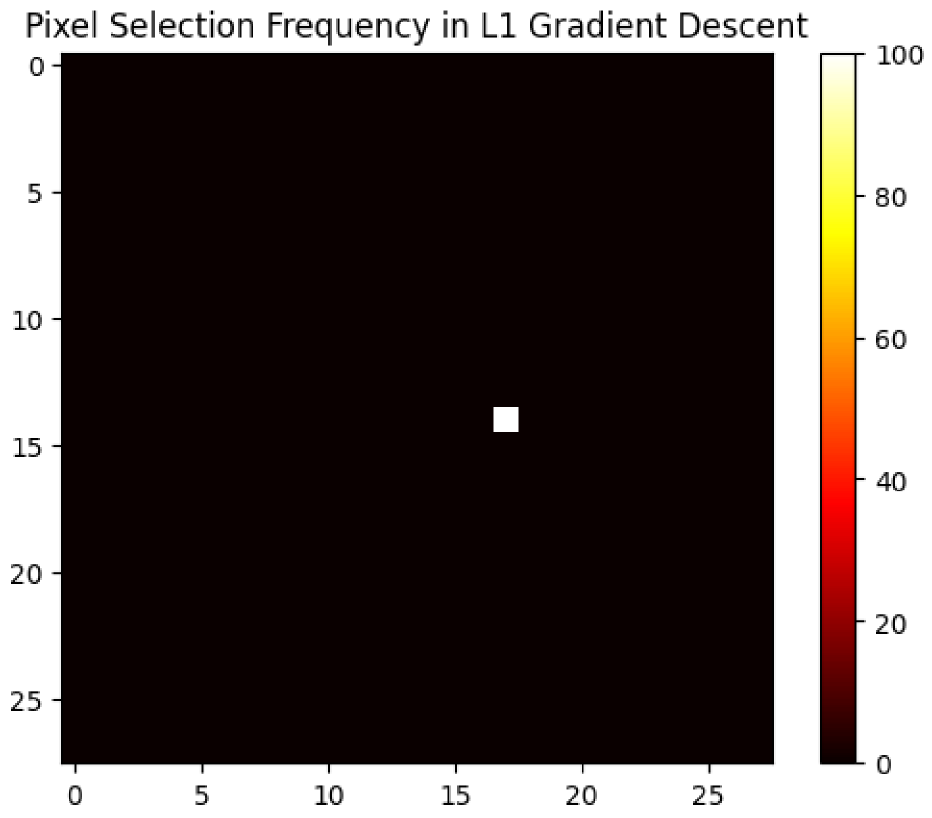
[51]:
```
plt.figure(figsize=(9, 5))
plt.plot(l2_losses_100, label="L2 Gradient Descent")
plt.plot(btls_losses_100, label="L2 BTLS Gradient Descent")
plt.plot(l_inf_losses_100, label="L∞ Gradient Descent")
plt.plot(l1_losses_100, label="L1 Gradient Descent (Coordinate)")
plt.plot(na_losses_100, label="Nesterov Accelerated Gradient Descent")
plt.plot(m_losses_100, label="Momentum Gradient Descent")
plt.xlabel("Iterations")
plt.ylabel("Loss")
plt.title("Loss over 100 Iterations for L∞, L1, L2, BTLS L2, Momentum, and N.A.␣
↪Gradient Descent")
plt.legend()
plt.show()
```



11 Reshape and plot the frequency counter as a 28x28 image:

Pixels chosen frequently (most grad) are **likely those that have the most impact on distinguishing between the digits 4 and 9**. The frequency map reveals which parts of the image space (such as specific strokes) are most informative for classification.

[52]:
```
frequency_image = frequency_counter.reshape(28, 28)
plt.imshow(frequency_image, cmap="hot", interpolation="nearest")
plt.colorbar()
plt.title("Pixel Selection Frequency in L1 Gradient Descent")
plt.show()
```



[ ]:
```
```