Building
Adadelta Optimizer
from Scratch in Python

Without relying on high-level libraries

Table of Contents

Building Adadelta Optimizer from Scratch in Python

1. Introduction

The Adadelta is an adaptive learning rate optimization algorithm designed to overcome the limitations of Adagrad, particularly its rapid decay of learning rates. Adadelta dynamically adapts over time using only the information of the gradients. It does not require a manually set learning rate and reduces the hyperparameter tuning process. This article will walk you through implementing the Adadelta optimizer from scratch in Python, explaining the fundamental concepts and each step involved.

What are Optimizer?

In deep learning, an optimizer is the algorithm responsible for updating the weights and biases of a neural network to minimize the loss function and improve its accuracy. **Think of it as the engine that drives the learning process.**

*** Cost Function (Loss Function)**: This function measures how far off the network's predictions are from the actual target values. The goal is to minimize this loss.
*** Learning Rate**: This parameter controls the size of the steps taken during each iteration. A smaller learning rate leads to slower but potentially more precise convergence, while a larger one might overshoot the minimum.
*** Gradients**: These are partial derivatives of the cost function with respect to the model parameters. They indicate the direction and rate of change of the cost function.
*** Weights and Biases**: These are the learnable parameters within the neural network that determine how the input data is transformed into predictions.
*** Optimizer's Role:** The optimizer uses the information from the loss function (specifically, its gradient) to adjust the weights and biases iteratively. It aims to find the optimal set of parameters that will result in the lowest possible loss.
* **Iterations/Epochs**: The number of times the entire training dataset is passed through the model.

Analogy: Imagine navigating a mountainous landscape in the dark, trying to find the lowest valley. The landscape represents the loss function, your current position is the current set of weights and biases, and the optimizer is your strategy for taking steps to descend efficiently.

Types of optimizer:

*** Gradient Descent Variants:** There are various algorithms like:
  * **Stochastic Gradient Descent (SGD):** Updates parameters based on the gradient calculated from a single data point (or a small batch) at a time.
  *** Batch Gradient Descent:** Calculates the gradient using the entire training dataset for each update.
  *** Mini-Batch Gradient Descent:** Strikes a balance by using a small batch of data points for each update.
*** Adaptive Optimizers:** Like Adam, RMSprop, and AdaGrad, these algorithms adjust the learning rate dynamically for each parameter based on its historical gradient information.

Choosing the right optimizer:

The optimal choice depends on the specific problem, dataset, and neural network architecture. Experimentation and understanding the strengths and weaknesses of different optimizers are crucial for achieving good performance in deep learning.

- **Start with Adam:** Adam is often a good default choice due to its robustness and generally good performance.
- **Consider SGD or Momentum:** For very large datasets or if you suspect Adam might be overfitting, SGD or Momentum with a well-tuned learning rate can be effective.
- **Experiment:** The optimal optimizer can depend on the specific problem, dataset, and neural network architecture. It's essential to experiment with different optimizers and hyperparameter settings to find what works best for your task.

2. What is Adadelta Optimizer?

Adadelta is an optimization algorithm used in machine learning for training models, particularly neural networks. It's designed to overcome some shortcomings of earlier methods like Adagrad, specifically the decaying learning rate problem.

Key Features:

- **Adaptive Learning Rates:** Like other adaptive optimizers (e.g., Adagrad, RMSprop), Adadelta adjusts the learning rate for each parameter individually throughout the training process. This allows it to take larger steps for infrequent parameters and smaller steps for frequent ones.
- **No Manual Learning Rate Setting:** Adadelta eliminates the need to manually tune the learning rate, a often tedious and time-consuming process.
- **Addresses Decaying Learning Rate:** Unlike Adagrad, which accumulates the squared gradients over all iterations (leading to a continuously diminishing learning rate), Adadelta considers only a fixed window of past gradients. This prevents the learning rate from becoming infinitesimally small.

Mathematical breakdown of the AdaGrad optimization algorithm

1. Exponential Moving Average (EMA):

- At the heart of Adadelta lies the concept of Exponential Moving Averages. Instead of directly using the current gradient or all past gradients, Adadelta maintains a "moving window" of past gradients using EMA.
- **Formula:**
  - $EMA_t = \rho * EMA_{t-1} + (1 - \rho) * Value_t$
  - Where:
    - EMA_t is the Exponential Moving Average at time t
    - ρ is the decay rate (a hyperparameter between 0 and 1)
    - $Value_t$ is the current value at time t
- **Intuition:** EMA gives more weight to recent values and exponentially less weight to older values. This helps Adadelta adapt to changing gradients during training.

2. Root Mean Square (RMS):

- RMS is used to calculate a "smoothed" representation of the magnitude of gradients and parameter updates.
- **Formula:**
  - $RMS = \sqrt{((Value_1^2 + Value_2^2 + ... + Value_n^2) / n)}$
- **Intuition:** RMS provides a measure of the "typical" size of the values, giving less weight to outliers.

3. Gradient Descent:

- Adadelta, like other optimization algorithms, is based on the fundamental principle of gradient descent. It aims to find the minimum of a loss function by iteratively updating parameters in the direction opposite to the gradient.
- **Basic Gradient Descent Update Rule:**
  - $\theta_{t+1} = \theta_t - \eta * \nabla J(\theta_t)$
  - Where:
    - θ_t is the parameter at time t
    - η is the learning rate
    - $\nabla J(\theta_t)$ is the gradient of the loss function J with respect to θ at time t

4. Adaptive Learning Rate:

- Unlike traditional gradient descent, which uses a fixed learning rate, Adadelta adjusts the learning rate for each parameter individually based on the history of gradients and updates.
- **Intuition:** This allows Adadelta to take larger steps in directions with consistently small gradients and smaller steps in directions with large or fluctuating gradients.

Combining the Concepts in Adadelta:

- Adadelta uses EMA to calculate moving averages of both the squared gradients ($E[g^2]$) and the squared parameter updates ($E[\Delta x^2]$).
- It then uses RMS to calculate a "smoothed" learning rate for each parameter based on these moving averages.
- This adaptive learning rate helps Adadelta converge faster and more smoothly compared to algorithms with a fixed learning rate.

How it Works:

Adadelta maintains two moving averages:

1. **Accumulated Squared Gradients ($E[g^2]$):** This is similar to Adagrad, but instead of summing over all past squared gradients, it uses an exponentially decaying average. This ensures that recent gradients have a higher impact than older ones.
2. **Accumulated Squared Parameter Updates ($E[\Delta x^2]$):** This tracks the history of squared parameter updates.

The Update Rule:

The parameter update rule for Adadelta is:

1. **Calculate the gradient (g) at the current time step.**
2. **Update the accumulated squared gradients ($E[g^2]$).**
3. **Calculate the root mean square (RMS) of the accumulated squared gradients.**
4. **Update the accumulated squared parameter updates ($E[\Delta x^2]$).**
5. **Calculate the RMS of the accumulated squared parameter updates.**
6. **Update the parameter (x) using the calculated RMS values.**

Advantages:

- **Robust to noisy gradients:** The exponentially decaying average smooths out the noise in gradient updates.
- **Faster convergence:** Compared to traditional gradient descent, Adadelta can converge faster, especially in scenarios with sparse data.
- **No manual learning rate tuning required:** This saves significant time and effort in hyperparameter optimization.

Disadvantages:

- **Can be computationally expensive:** Maintaining the moving averages adds computational overhead compared to simpler optimizers.
- **May not always outperform other adaptive methods:** The performance of different optimizers can vary depending on the specific dataset and model architecture.

In Summary:

Adadelta is a powerful optimization algorithm that offers several advantages over traditional gradient descent and even other adaptive methods like Adagrad. It's particularly well-suited for problems with sparse data and can significantly speed up the training process. However, it's essential to consider its computational cost and experiment with other optimizers to determine the best choice for your specific machine learning task.

3. Comparison of popular adaptive optimization algorithms

Feature	Adagrad	Adadelta	RMSprop	Adam
Basic Idea	Accumulates squared gradients to adapt learning rates.	Uses moving averages of both squared gradients and parameter updates.	Similar to Adadelta, using a moving average of squared gradients.	Combines momentum with adaptive learning rates.
Learning Rate	Adaptively decreases for each parameter.	Adaptively adjusts based on history of gradients and updates.	Adaptively adjusts based on the history of gradients.	Adaptively adjusts for each parameter with momentum.
Momentum	No	No	No	Yes
Addressing Adagrad's Decaying Learning Rate	No	Yes, uses a moving average window.	Yes, uses a moving average window.	Yes, indirectly by incorporating momentum.
Hyperparameters	Learning rate (η), epsilon (ε)	Decay rate (ρ), epsilon (ε)	Decay rate (ρ), learning rate (η), epsilon (ε)	Learning rate (η), decay rates (β1, β2), epsilon (ε)
Computational Cost	Low	Moderate	Moderate	Moderate
Advantages	Simple, works well for sparse data	No manual learning rate tuning, robust to noisy gradients	Often faster convergence than Adagrad	Combines benefits of adaptive learning rates and momentum
Disadvantages	Learning rate can decay too aggressively	Can be sensitive to the choice of decay rate	Can sometimes be outperformed by Adam	More hyperparameters to tune

Summary:

- **Adagrad:** A good starting point, but its aggressively decaying learning rate can hinder convergence in later stages of training.
- **Adadelta:** Solves the decaying learning rate issue of Adagrad and eliminates the need for manual learning rate tuning.
- **RMSprop:** Very similar to Adadelta, often considered a simpler alternative with slightly better empirical performance in some cases.
- **Adam:** Combines the benefits of adaptive learning rates with momentum, often leading to faster convergence and better overall performance. It's generally a good default choice for many optimization tasks.

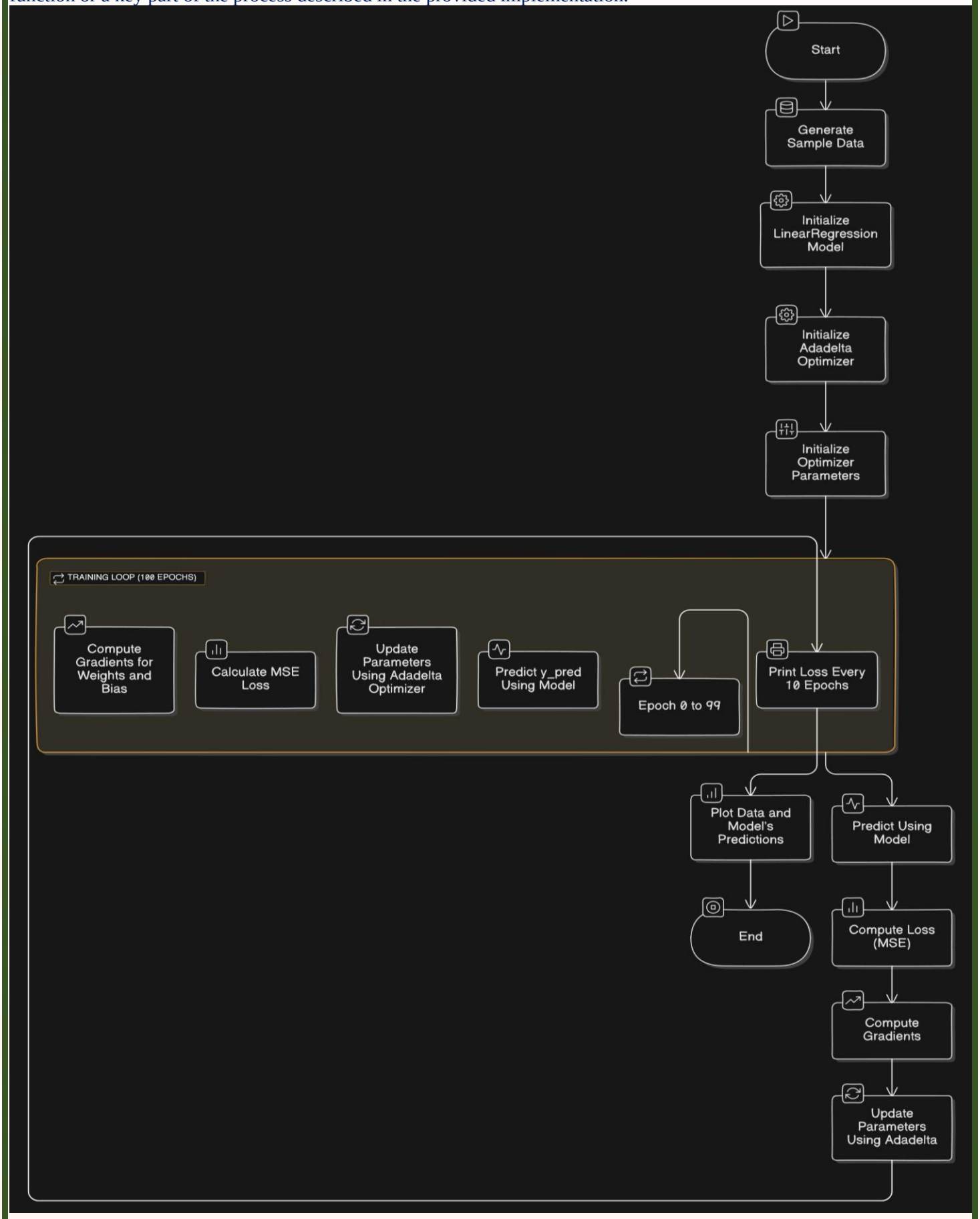
Choosing the Right Algorithm:

- **Start with Adam:** It's often a strong performer and a good default choice.
- **Consider RMSprop:** If Adam doesn't perform well or if you want a simpler algorithm.
- **Try Adadelta:** If you want to avoid manual learning rate tuning and your data is particularly noisy.
- **Adagrad can be suitable:** For specific situations with sparse data where its aggressive learning rate decay is beneficial.

It's important to experiment with different optimizers and hyperparameter settings to find the best approach for your specific machine learning problem and dataset.

4. The Structure of a Gradient Descent Optimizer

This Structure includes the steps and sub-steps with appropriate labels and connections. Each step corresponds to a function or a key part of the process described in the provided implementation.



Building Adadelta Optimizer from Scratch in Python

5. Implementation in Python

Let's implement a Adadelta Optimizer in Python.

Step 1: Initialize Parameters

We import numpy for numerical computations and matplotlib.pyplot for plotting graphs. First, we need to initialize the parameters for our Adadelta optimizer, including the decay rate (ρ), epsilon (ϵ), and the running averages for squared gradients and squared updates.

```
import numpy as np

class Adadelta:
    def __init__(self, rho=0.95, epsilon=1e-6):
        self.rho = rho
        self.epsilon = epsilon
        self.Eg2 = None
        self.Edx2 = None

    def initialize(self, params):
        # Initialize running averages for squared gradients and updates
        self.Eg2 = [np.zeros_like(p) for p in params]
        self.Edx2 = [np.zeros_like(p) for p in params]
```

Step 2: Compute Gradient Updates

we'll compute the gradient updates for each parameter using the Adadelta equations.

```
 def update(self, params, grads):
     updates = []
     for i, (p, g) in enumerate(zip(params, grads)):
         # Update running average of squared gradients
         self.Eg2[i] = self.rho * self.Eg2[i] + (1 - self.rho) * g ** 2

         # Compute parameter update
         update = - (np.sqrt(self.Edx2[i] + self.epsilon) / np.sqrt(self.Eg2[i] + self.epsilon)) * g

         # Update running average of squared updates
         self.Edx2[i] = self.rho * self.Edx2[i] + (1 - self.rho) * update ** 2

         # Apply the update
         p += update
         updates.append(update)
     return updates
```

Building Adadelta Optimizer from Scratch in Python

Step 3: Integrate with a Model

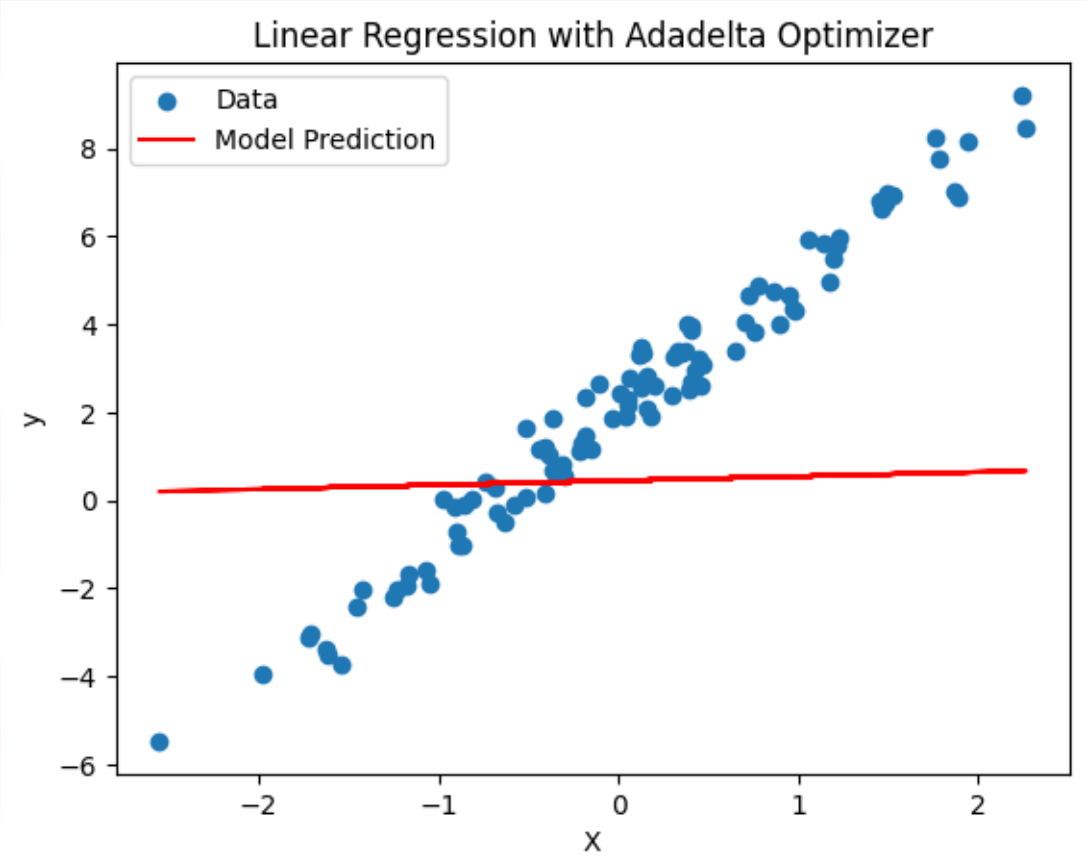
To demonstrate the use of our Adadelta optimizer, let's create a simple linear regression model and integrate the optimizer with it.

```
class LinearRegression:
    def __init__(self, input_dim):
        # Initialize model parameters (weights and bias)
        self.W = np.random.randn(input_dim, 1)
        self.b = np.zeros((1,))

    def predict(self, X):
        # Linear prediction
        return np.dot(X, self.W) + self.b

    def compute_gradients(self, X, y, y_pred):
        # Compute gradients for weights and bias
        m = X.shape[0]
        dW = np.dot(X.T, (y_pred - y)) / m
        db = np.sum(y_pred - y) / m
        return [dW, db]

# Generate some sample data for demonstration
np.random.seed(0)
X = np.random.randn(100, 1)
y = 3 * X.squeeze() + 2 + np.random.randn(100) * 0.5
y = y.reshape(-1, 1)

# Initialize model and optimizer
model = LinearRegression(input_dim=1)
optimizer = Adadelta()
optimizer.initialize([model.W, model.b])

# Training loop
num_epochs = 100
for epoch in range(num_epochs):
    # Forward pass
    y_pred = model.predict(X)

    # Compute loss (Mean Squared Error)
    loss = np.mean((y_pred - y) ** 2)

    # Backward pass
    grads = model.compute_gradients(X, y, y_pred)

    # Update parameters using Adadelta
    optimizer.update([model.W, model.b], grads)

    # Print loss every 10 epochs
    if epoch % 10 == 0:
        print(f'Epoch {epoch}, Loss: {loss}')
```

Step 4: Visualize Results

let's visualize the training process and the model's predictions.

```
import matplotlib.pyplot as plt

# Plot the data and the model's predictions
plt.scatter(X, y, label='Data')
plt.plot(X, model.predict(X), color='red', label='Model Prediction')
plt.xlabel('X')
plt.ylabel('y')
plt.legend()
plt.title('Linear Regression with Adadelta Optimizer')
plt.show()
```



6. Conclusion

In this post, we implemented the Adadelta optimizer from scratch in Python. We explored the fundamental concepts, step-by-step implementation, and integration with a simple linear regression model. Adadelta is a powerful optimization algorithm that adapts the learning rate dynamically, making it a valuable tool for training machine learning models.

By understanding and implementing such optimizers from scratch, you gain deeper insights into their working mechanisms and can better tune and debug machine learning models.

Constructive comments and feedback are welcomed