Here's the corrected version of the README file content for "Gradient Descent and Its Variants":

Gradient Descent and Its Variants

1. Gradient Descent (GD)

Definition:
Gradient Descent is an iterative optimization algorithm used to minimize a differentiable function by iteratively moving towards the steepest descent direction, which is defined by the negative of the gradient.

Formula:

$$\theta = \theta - \eta \nabla J(\theta)$$

where:
-
$$\theta$$
represents the parameters,
-
$$\eta$$
is the learning rate,
-
$$\nabla J(\theta)$$
is the gradient of the cost function
$$J$$
.

Purpose:
To minimize the cost function in machine learning models, thereby improving their accuracy.

When to Use:
Use GD when you have a convex function and can afford to compute gradients for the entire dataset.

Advantages: - Stable convergence for convex functions. - Simple to implement.

Disadvantages: - Can be slow for large datasets as it requires computing gradients for all data points. - May converge to local minima in non-convex problems[1][4].

---

2. Stochastic Gradient Descent (SGD)

Definition:
Stochastic Gradient Descent updates parameters using only one training example at a time.

Formula:

$$\theta = \theta - \eta \nabla J_i(\theta)$$

where

$$J_i$$

is the cost for the

$$i^{th}$$

training example.

Purpose:
To provide faster updates and facilitate learning from large datasets.

When to Use:
Use SGD when dealing with large datasets where full batch updates are computationally expensive.

Advantages: - Faster convergence due to frequent updates. - Helps escape local minima due to noise in gradient estimates.

Disadvantages: - Noisy updates can lead to oscillations around the minimum. - May require careful tuning of learning rates[2][4].

---

3. Mini-Batch Stochastic Gradient Descent

Definition:
This variant combines features of both batch and stochastic gradient descent by updating parameters using a small random subset (mini-batch) of the training data.

Formula:

$$\theta = \theta - \eta \nabla J_{mini-batch}(\theta)$$

Purpose:
To balance between the convergence speed of SGD and the stability of Batch Gradient Descent.

When to Use:
Use mini-batch SGD when you want faster convergence without sacrificing too much stability.

Advantages: - Reduces variance in parameter updates, leading to more stable convergence. - Efficient use of computational resources by leveraging vectorized operations on mini-batches.

Disadvantages: - Still requires tuning of mini-batch size. - May not fully eliminate noise from stochastic updates[3][4].

---

4. SGD with Momentum

Definition:
This method adds a momentum term that helps accelerate gradient vectors in the right direction, leading to faster convergence.

Formula:

$$v = \beta v + (1 - \beta)\nabla J(\theta)$$

$$\theta = \theta - \eta v$$

where

$$v$$

is the velocity vector and

$$\beta$$

is the momentum coefficient.

Purpose:
To improve convergence speed and reduce oscillation by accumulating past gradients.

When to Use:
Use when gradients are noisy or when faster convergence is needed in deep learning models.

Advantages: - Accelerates convergence in relevant directions. - Helps overcome local minima and saddle points.

Disadvantages: - Requires careful tuning of the momentum coefficient. - Can overshoot if not properly configured[1][2].

---

5. Adagrad

Definition:
Adagrad adapts the learning rate for each parameter individually based on past gradients, allowing larger updates for infrequent features and smaller updates for frequent ones.

Formula:

$$\theta = \theta - \frac{\eta}{\sqrt{G_t + \epsilon}} \nabla J(\theta)$$

where

$$G_t$$

is the sum of squares of past gradients.

Purpose:
To automatically adjust learning rates based on parameter updates, improving performance on sparse data.

When to Use:
Use Adagrad when working with sparse datasets or features.

Advantages: - No need for manual tuning of learning rates. - Works well with sparse data representations.

Disadvantages: - Learning rate can become excessively small, leading to premature convergence[3][4].

---

6. RMSProp

Definition:
RMSProp modifies Adagrad by using a moving average of squared gradients to normalize the gradient, preventing rapid decay of learning rates.

Formula:

$$G_t = \beta G_{t-1} + (1 - \beta)(\nabla J(\theta))^2$$

$$\theta = \theta - \frac{\eta}{\sqrt{G_t + \epsilon}} \nabla J(\theta)$$

Purpose:
To maintain a more stable learning rate throughout training, especially in non-stationary settings.

When to Use:
Use RMSProp in scenarios with noisy or non-stationary objectives.

Advantages: - Prevents diminishing learning rates seen in Adagrad. - Effective in training deep networks with varying feature distributions.

Disadvantages: - Requires tuning of the decay rate parameter. - Still sensitive to initial learning rate settings[1][3].

---

7. Adam Optimizer

Definition:
Adam (Adaptive Moment Estimation) combines ideas from RMSProp and momentum by maintaining both an exponentially decaying average of past gradients and squared gradients.

Formula: 1. Update biased first moment estimate:

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$

2. Update biased second moment estimate:

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$

3. Correct bias in first moment estimate:

$$m'_t = \frac{m_t}{1 - \beta_1^t}$$

4. Correct bias in second moment estimate:

$$v'_t = \frac{v_t}{1 - \beta_2^t}$$

5. Update parameters:

$$\theta = \theta - \frac{\eta}{\sqrt{v'_t} + \epsilon} m'_t$$

where

$$g_t$$

is the gradient at time step

$$t$$

.

Purpose:
To combine the advantages of both momentum and adaptive learning rates for improved performance across various datasets and models.

When to Use:
Adam is generally effective across a wide range of problems, particularly deep learning tasks where training dynamics can be complex.

Advantages: - Efficient computation with low memory requirements. - Generally performs well without extensive hyperparameter tuning.

Disadvantages: - Can sometimes lead to suboptimal solutions compared to other optimizers if not tuned properly.
