YILDIZ TEKNİK ÜNİVERSİTESİ
ELEKTRİK-ELEKTRONİK FAKÜLTESİ
BİLGİSAYAR MÜHENDİSLİĞİ BÖLÜMÜ
BİLGİSAYAR MÜHENDİSLERİ İÇİN DİFERANSİYEL DENKLEMLER DERSİ
BİRİNCİ ÖDEV

MNIST Veri Seti ile Temel Makine Öğrenimi Algoritmalarının Karşılaştırılması

https://github.com/mertgulerx/mnist-algorithm-comparison-plain-c

İÇİNDEKİLER

GİRİŞ

Bu projede, MNIST veri seti üzerinde üç farklı temel optimizasyon algoritması olan Gradient Descent (GD), Stochastic Gradient Descent (SGD) ve ADAM kullanılarak makine öğrenimi modelinin eğitimi ve testleri yapılacaktır. Model temel şekilde tasarlanmış olup herhangi bir gizli katman içermemektedir. SGD ve ADAM için batch size methotları kullanılmıştır.

Amaç

Bu projedeki temel amaç, farklı optimizasyon algoritmalarının performanslarını karşılaştırmak, bu algoritmaların çeşitli değişkenlerden ne şekilde etkilendiğini gözlemlemek, doğruluk, kayıp ve süre gibi kriterleri göz önüne alarak veri seti için en verimli olanını belirlemektir.

YÖNTEM

Kullanılan Veri Seti

El yazısı rakamlarının görüntüsünü içeren MNIST veri setinin özel bir versiyonu kullanılmıştır. Veri seti, 28x28 boyutlarında, 0 ile 9 arasındaki rakamları içeren toplam 42.000 eğitim ve test görüntüsünden oluşmaktadır. Bu görüntülerdeki her bir piksel değeri, gri tonlamada 0 ile 255 arasında bir değer alır.

Veri İşleme

Veriler ".csv" dosya uzantılı bir veri setinden alınmaktadır. Bu dosya 785 satır ve 42001 sütün içermektedir. Bunlardan biri "etiket" değeri olup görüntünün hangi sayıya ait olduğunu içerir. Geri kalanlar ise görüntülerin pixel değerleridir. Görüntüler veri setinde rastgele dağıtılmıştır.

İkili sınıflandırma yapabilmek için 0-9 arasındaki rakamlardan sadece 0 ve 1 rakamlarının görüntüleri alınmıştır. Görüntüler alındıktan sonra 28x28 boyutundan 784 + 1 boyutlu bir vektöre dönüştürülür. Buradaki ekstra sapma (bias) değeridir. Sapma değeri 1 olarak girilmiştir.

Veri, modelin eğitimine başlamadan önce normalize edilmiştir. Pixel değerleri 0 ile 1 aralığına çekilmiştir.

Ağ Yapısı

Bu modelde basit bir yapay ağ kullanılmıştır. Herhangi bir gizli ağ bulunmamaktadır. Yalnızca giriş ve çıkış katmanlarını içerir.

Giriş katmanı, veri setindeki 785 giriş nöronuna sahiptir. Çıkış katmanı ise 2 adet nöron içerir.

Projenin aktivasyonu için **tanh** fonksiyonu kullanılmıştır.

Kullanılan Diller ve Kütüphaneler
Projede, modeli tasarlamak için C programlama dili kullanılmıştır. Kütüphane olarak standart kütüphaneler kullanılmıştır.
Grafikleri tasarlamak için Python programlama dili kullanılmıştır. Matplotlib, pandas, scikit-learn kütüphaneleri kullanılmıştır.

Algoritmalar
Üç farklı temel optimizasyon algoritması kullanılmıştır:

Gradient Descent (GD), model parametrelerini güncellemek için tüm eğitim veri setini kullanan bir optimizasyon algoritmasıdır. Her adımda, kayıp fonksiyonunun gradyanını hesaplayarak parametreleri günceller.

Stochastic Gradient Descent (SGD), Gradient Descent'in daha hızlı bir versiyonudur. Her bir güncellemede, tüm veri seti yerine yalnızca rastgele seçilen bir veya bir batch (parti) boyutunda örnek kullanır. Projede bu batch boyutlarının performansa etkisinin karşılaştırması yapılmıştır.

ADAM, optimizasyon için sık kullanılan modern bir algoritmadır. Gradient Descent ve SGD'den farklı olarak, öğrenme oranını dinamik bir şekilde ayarlayarak öğrenme yapar. SGD'de olduğu gibi bu algoritmada da batch kullanılmıştır.

Kayıp Fonksiyonu ve Doğruluk Hesabı
Başarı oranını ölçmek için Ortalama Kare Hatası (Mean Squared Error, MSE) kullanılmıştır. Doğruluk oranını bulabilmek için GD algoritmasında hata oranı, eğitim örneği sayısına, SGD ve ADAM algoritmaları için ise batch boyutuna bölünmüştür.

Ağırlık Değerlerinin Başlatılması
Ağırlık değerleri her bir pixel için rastgele olarak başlatıldı. Bu seçimler belli bir aralık değerleriyle beş farklı kez yapıldı.

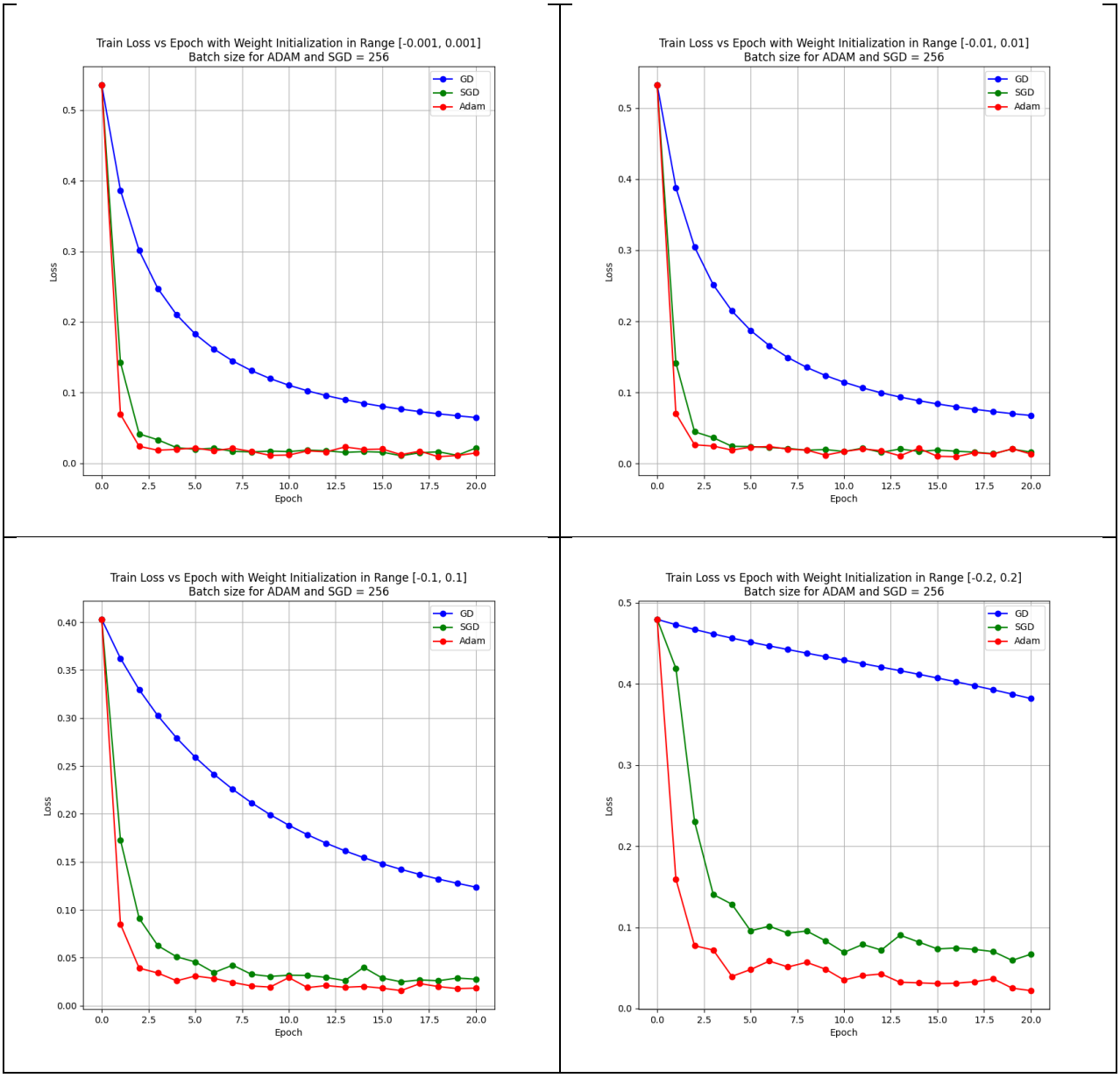
DENEYLER VE SONUÇLAR

Deney sonuçları grafikler şeklinde verilmiştir. Dosya ve sayfa kenarlıkları limitasyonları sebebiyle örnek sayısı 4'e düşürülmüş olup kalan örnekler, ekler sayfasında yer almaktadır.

Grafiklerde ağırlıkların başlatılma aralıkları {0.001, 0.005, 0.01, 0.1, 0.2} olarak artacak şekilde ayarlanmıştır. (0.005 grafikleri ekler kısmında bulunmaktadır.)

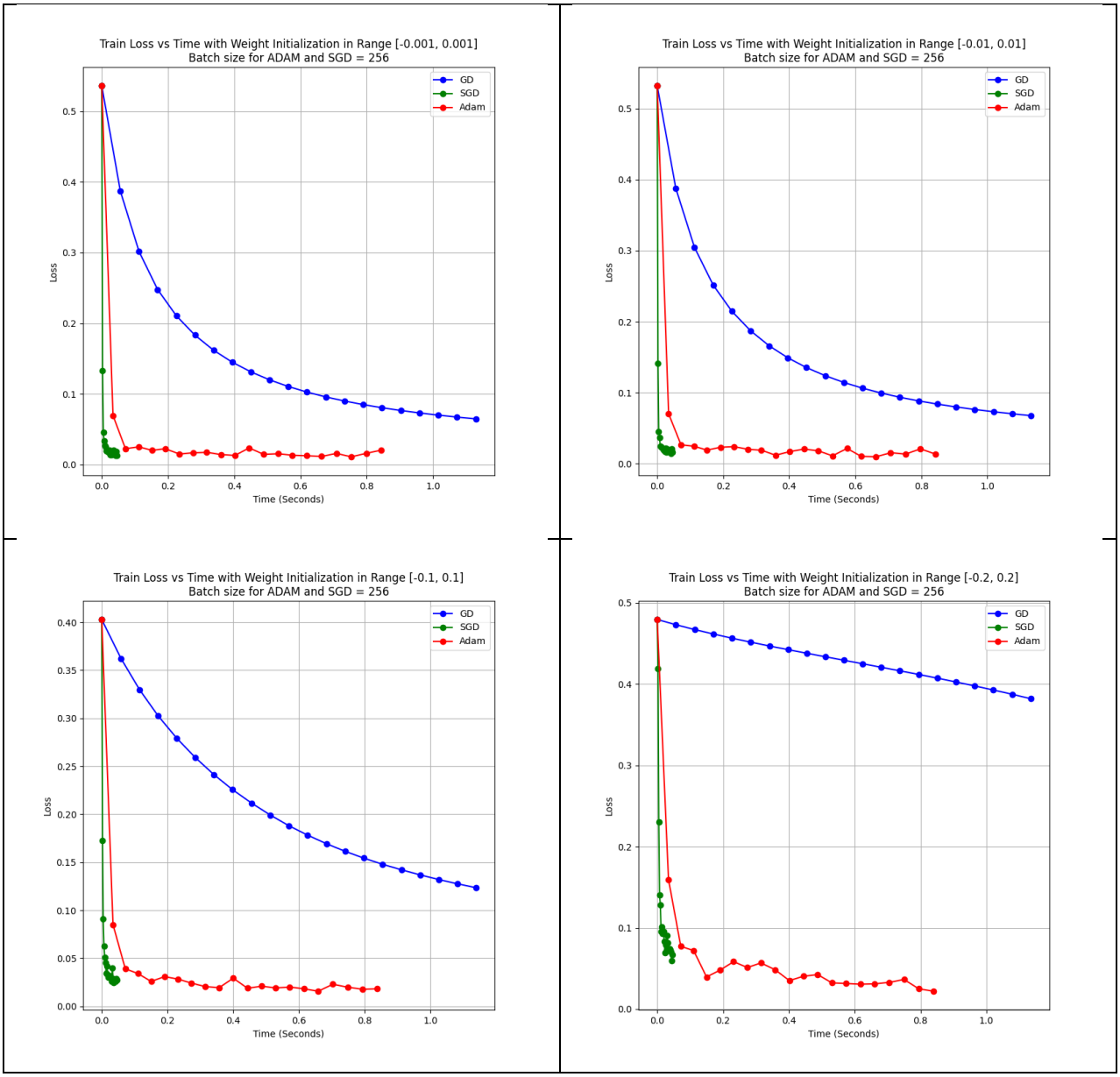
Eğitim Sonuçları

Kayıp / Güncelleme Sayısı Grafikleri



Yukarıdaki sonuçlar incelendiğinde SGD ve ADAM algoritmalarının çok hızlı bir şekilde düşük kayıp değerlerine ulaştığı gözlemlenmiştir. GD ise dört örnekte de diğer iki algoritmanın kayıp değerlerine ulaşamamıştır. Daha fazla güncelleme sayısına ihtiyaç duymaktadır. SGD ve ADAM algoritmalarında algoritmaların yapısı gereği dalgalanmalar gözlendi. GD ise sabit bir şekilde ilerledi. Ağırlık aralıklarının boyutu arttıkça her üç algoritmanın da performansının düştüğü, bu düşüşlerden en az ADAM etkilendi. GD ise çok büyük bir düşüş yaşadı. Artan ağırlık sınırlarıyla beraber SGD ve ADAM algoritmalarının dalgalanma boyutları arttı. SGD ve ADAM algoritmalarında bir miktar overfitting bulunmaktadır.

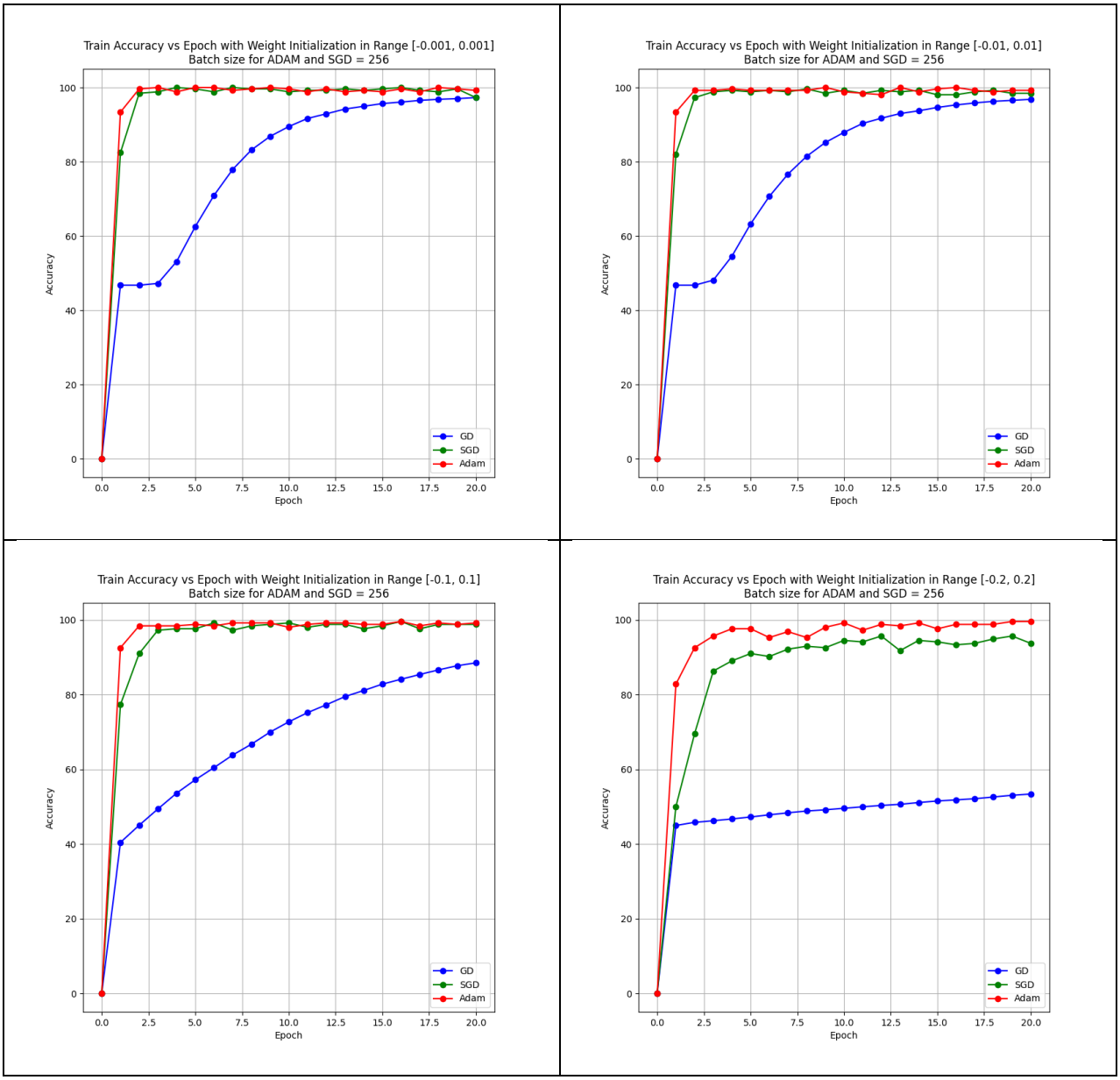
Kayıp / Zaman Grafikleri



Algoritmalar kayıp / zaman bakımından incelendiğinde SGD'nin büyük bir farkla en hızlı olduğu, onu sırasıyla ADAM ve GD'nin takip ettiği bulundu. Bunun sebebi ise her epochta GD'nin tüm örnekler üzerinden ağırlık güncellemesi yapması (Yaklaşık 7000 kez), SGD ve

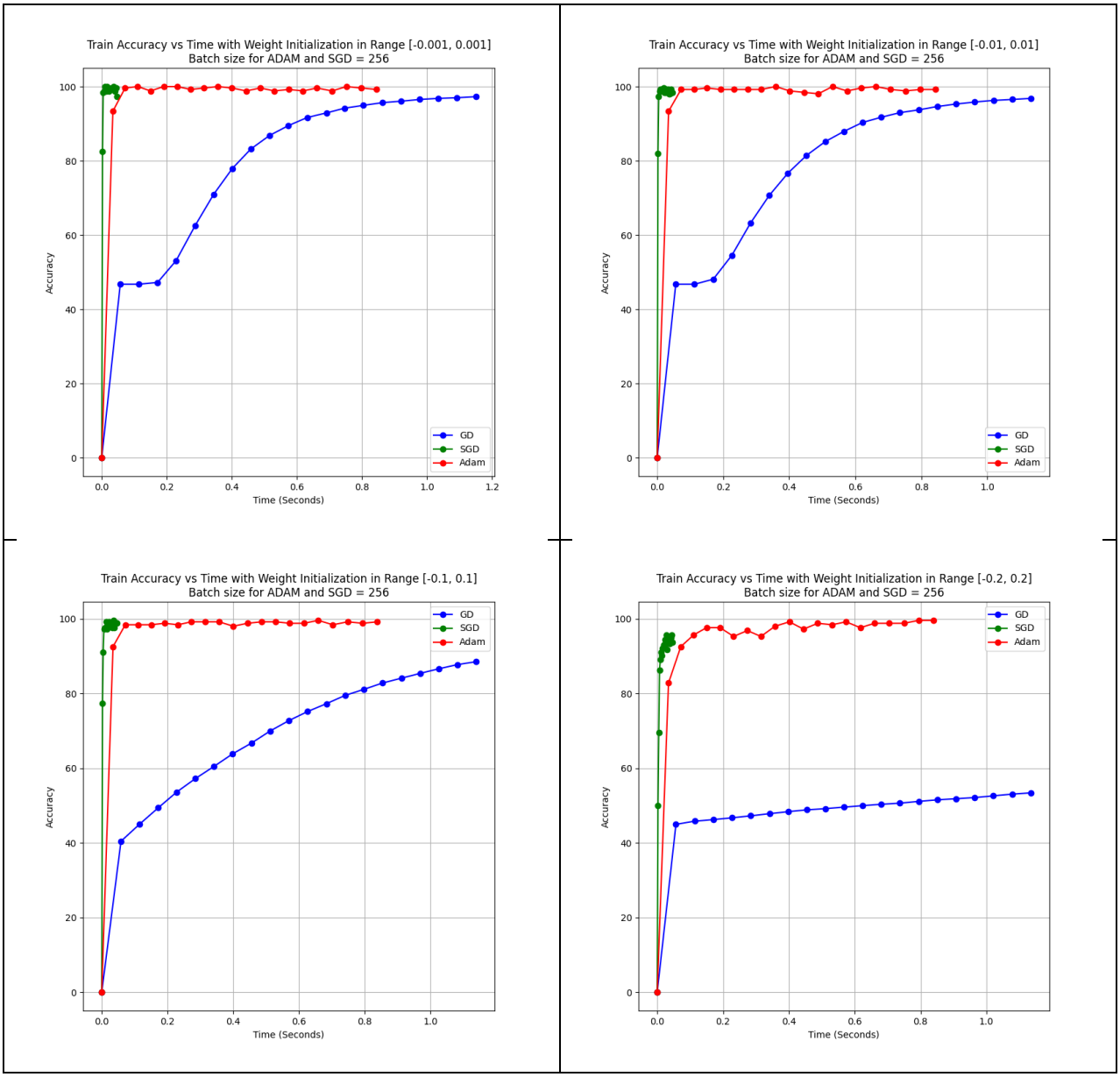
ADAM'ın ise nin belirtilen batch boyutu kadar yani 256 kez güncelleme yapmasıdır. ADAM adaptif öğrenme hızı uygulamak için daha fazla hesap yaptığından dolayı SGD'den daha yavaştır. Batch boyutlarının hıza etkisi bir çarpımdır. Genel olarak 16 ila 256 arasındaki değerler kullanılır. 32 boyutunda bir batch kullanıldığını varsaysaydık bu zaman değerleri 5 kat daha az çıkacaktı. Grafikte detaylı inceleme için boyut 256 seçilmiştir. Kayıp değerlerinin analizi güncelleme sayısı grafikleriyle aynıdır.

Doğruluk / Güncelleme Sayısı Grafikleri



Analizler diğer grafiklerle aynıdır. Doğruluk değerlerinin hesabı Yöntemler sayfasında belirtildiği gibidir. 0,001 ve 0,01 ağırlık sınırlarında GD'nin doğruluğunun SGD ve ADAM'a çok yaklaştığı gözlemlendi. SGD ve GD 0,2 ağırlık sınırında büyük düşüş yaşadı. ADAM ise doğruluk değerini korudu. SGD kayıp grafiklerinin aksine 0,1 aralığında doğruluk değerlerinde düşüş yaşamadı.

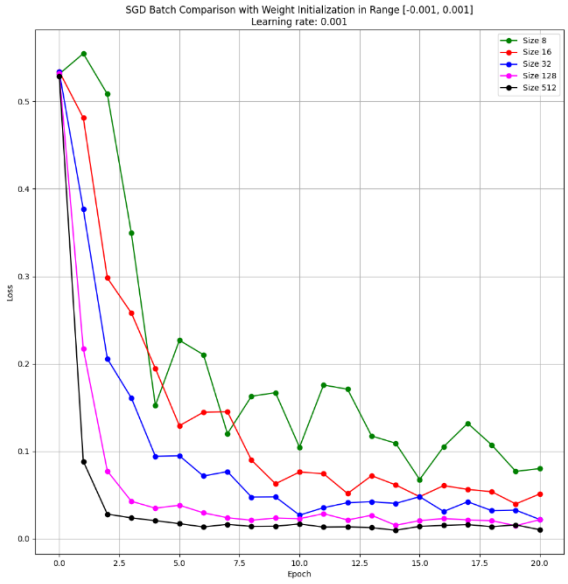
Doğruluk / Zaman Grafikleri



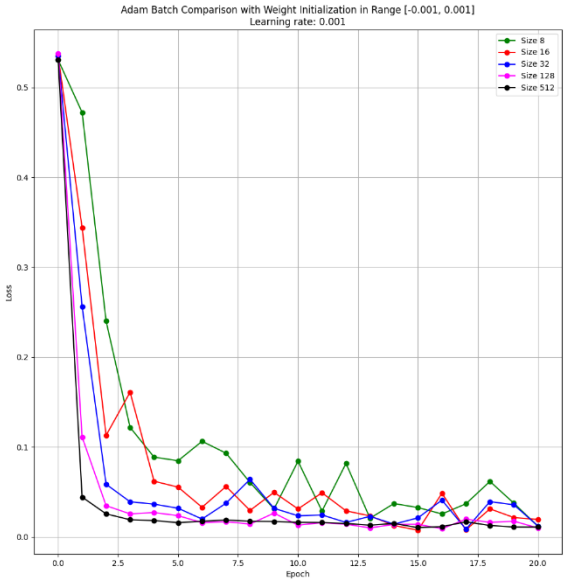
Analizler diğer grafikler gibidir. SGD ve GD artan ağırlık sınırlarından etkilenirken ADAM çok az miktarda etkilendi. Hız sıralaması SGD > ADAM > GD şeklinde oldu.ADAM, adaptif öğrenme oranı sayesinde doğruluğunu korudu.

Batch Boyutu Karşılaştırması

SGD ve ADAM için örnekleri belirli bir parti (batch) boyutunda rastgele şekilde seçilmekteydi. Bu parti boyutunu arttırmak hem zaman hem de doğruluk bakımından modeli etkiliyor. Parti boyutu zamanı çarpım şeklinde etkilemektedir. Doğrulu ise aşağıdaki grafiklerdeki gibi etkilemektedir. Parti boyutları {8, 16, 32, 128, 512} şeklinde ayarlanmıştır.



SGD için parti boyutu karşılaştırması



ADAM için parti boyutu karşılaştırması

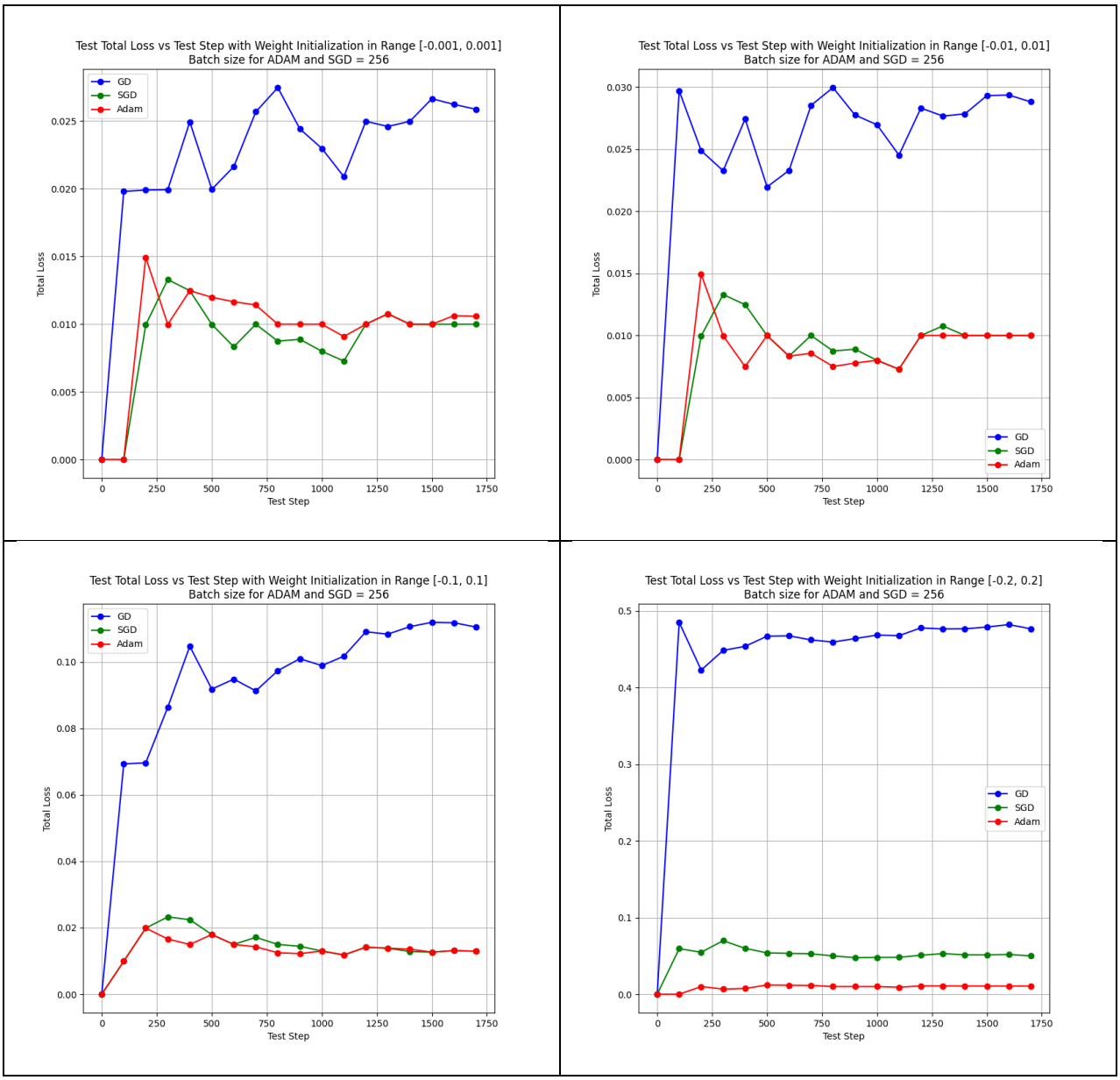
Grafikler incelendiğinde parti boyutunun artması ile her iki algoritma için de dalgalanmalarda azalma gözlenmiştir ve kayıp oranı azalmıştır. Dolaylı olarak da doğruluk oranı artmıştır. SGD parti boyutu değişimlerinden yüksek miktarda etkilenmiştir. Adam ise daha az etkilenmiştir. SGD, 128 parti boyutunda; ADAM ise 32 parti boyutunda yeterli kararlılığa ulaşmıştır. 512 parti boyutunda her iki algoritma da çok iyi sonuç verdi. Fakat zaman verimliliği açısından bu parti değerlerinden en iyi değerin 128 olduğunu söyleyebiliriz.

Test Sonuçları

Test sonuçları incelemesi yapılırken bazı noktalar eğitim sonuçlarından farklı olarak değerlendirilmelidir. Öncelikle her üç algoritma için de süre farkı bulunmamaktadır. Grafikler çizilirken bazı limitasyonlar mevcuttur. Bunlar:

Kayıp (loss) değerleri her adım için doğru hesaplanamaz. Bu yüzden grafiklerde her adım için kayıp değeri yerine toplam kayıp değeri kullanıldı. Dalgalanmalar grafik hakkında kesin bir bilgi vermezken değerlerin ortalamasına ve bitişlerine bakarak daha doğru yorum yapılabilir. Test sonuçları 20 epoch ile eğitim sonrasında alınmıştır.

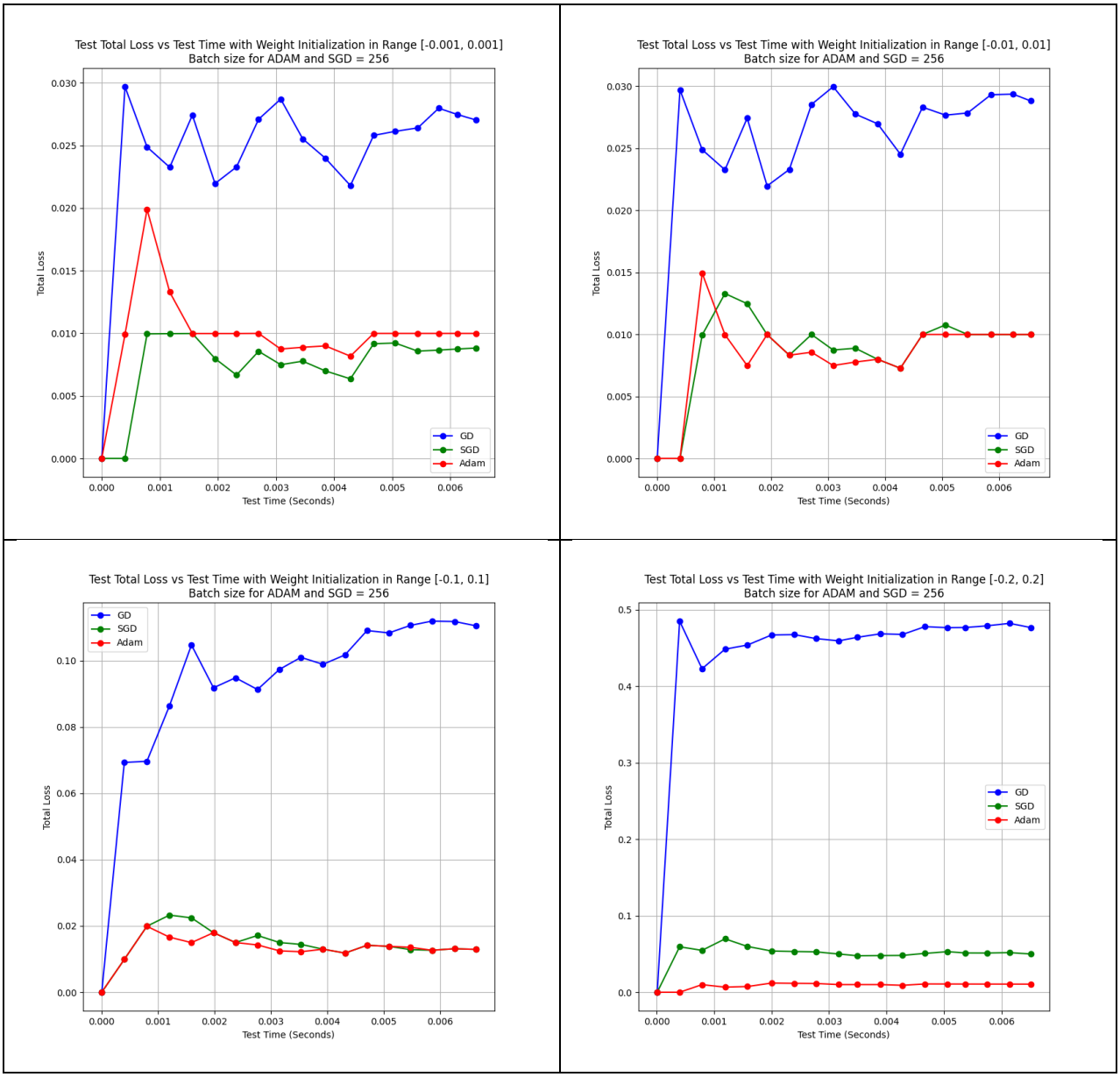
Toplam Kayıp / Güncelleme Sayısı Grafikleri



Test sonuçlarını incelediğimizde:
Doğruluk açısından SGD ve ADAM'ın aşırı yakın olduğunu, fakat ADAM'ın her zaman bir miktar daha iyi olduğunu görebiliriz. Bir kazanan seçmek istersek zaman verimliliği açısından galibimiz kesinlikle SGD olmalı. Ağırlık sınırlarını 0.2 yaptığımızda ADAM çok daha iyi sonuç verse de optimize hali yani 0.01 sınırını ele almalıyız.

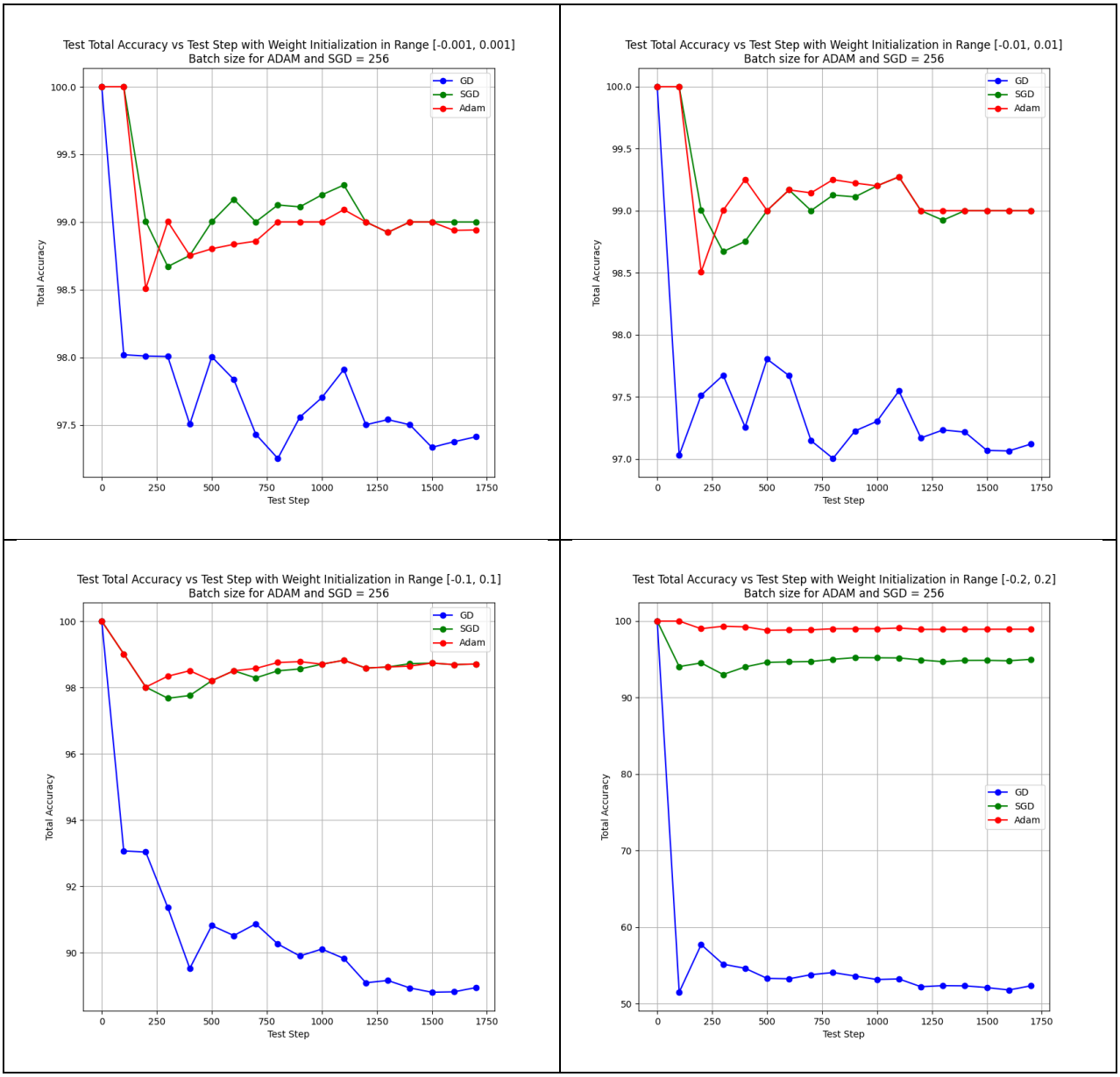
GD ise diğer iki algoritmanın performansına yaklaşamamıştır. Fakat doğruluk değerleri arasındaki fark ilk üç grafikte çok azdır. Ortalama %2-4 aralığında bir doğruluk farkı mevcuttur. Bu da GD'nin çok kötü bir performans vermediğine işaret ediyor.

Toplam Kayıp / Zaman Grafikleri



Zaman grafiklerini incelediğimizde SGD ve ADAM'ın tekrardan çok yakın olduğunu görebiliz. ADAM daha kötü bir başlangıç yapsa da SGD'den çok daha kararlı bir test izliyor. İlk grafikte SGD öne geçse de diğer grafiklerde ADAM hem daha stabil şekilde izliyor hem de daha az veya yakın bir kayıp değeri sunuyor.

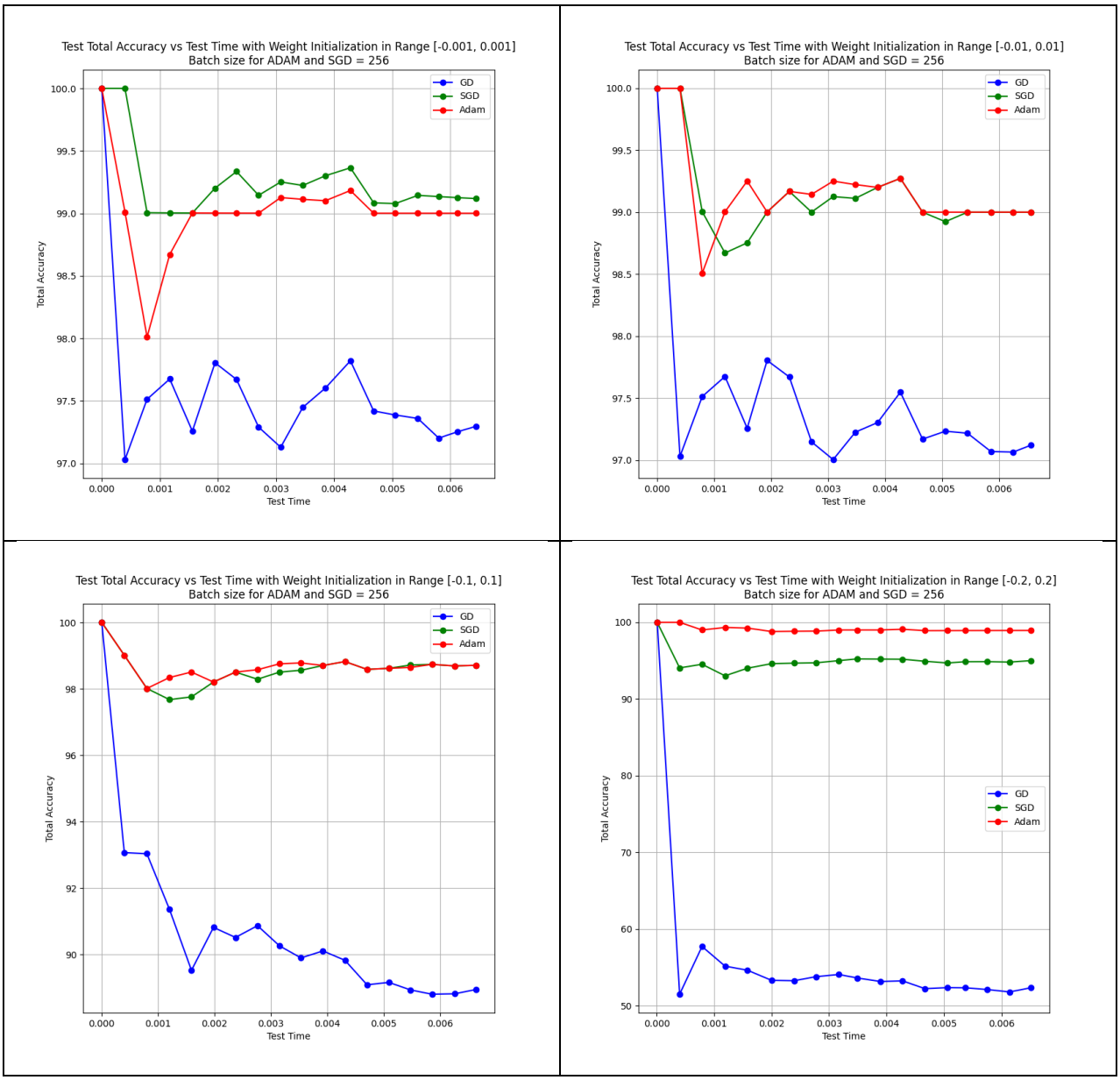
Toplam Doğruluk / Güncelleme Sayısı Grafikleri



Doğruluk değerlerini incelediğimizde üç algoritma da çok iyi sonuç vermiş. 0.01 ağırlık aralığını baz aldığımızda test sonucunda:
GD, %97.2
SGD, %99.02
ADAM; %99.11

Doğruluk değeri almıştır. Bu üç algoritmanın da iyi bir şekilde çalıştığına işaret ediyor. 20 epoch gibi düşük bir değer için oldukça iyi.
ADAM, her zamanki gibi stabil bir sonuç veriyor. SGD'nin %0.09 gibi bir farkla geride kalması ve iki algoritmanın eğitim süreleri arasındaki farkı değerlendirdiğimizde söylemeliyiz ki SGD kesinlikle verimlilik bakımından bu veri seti için çok daha önde.

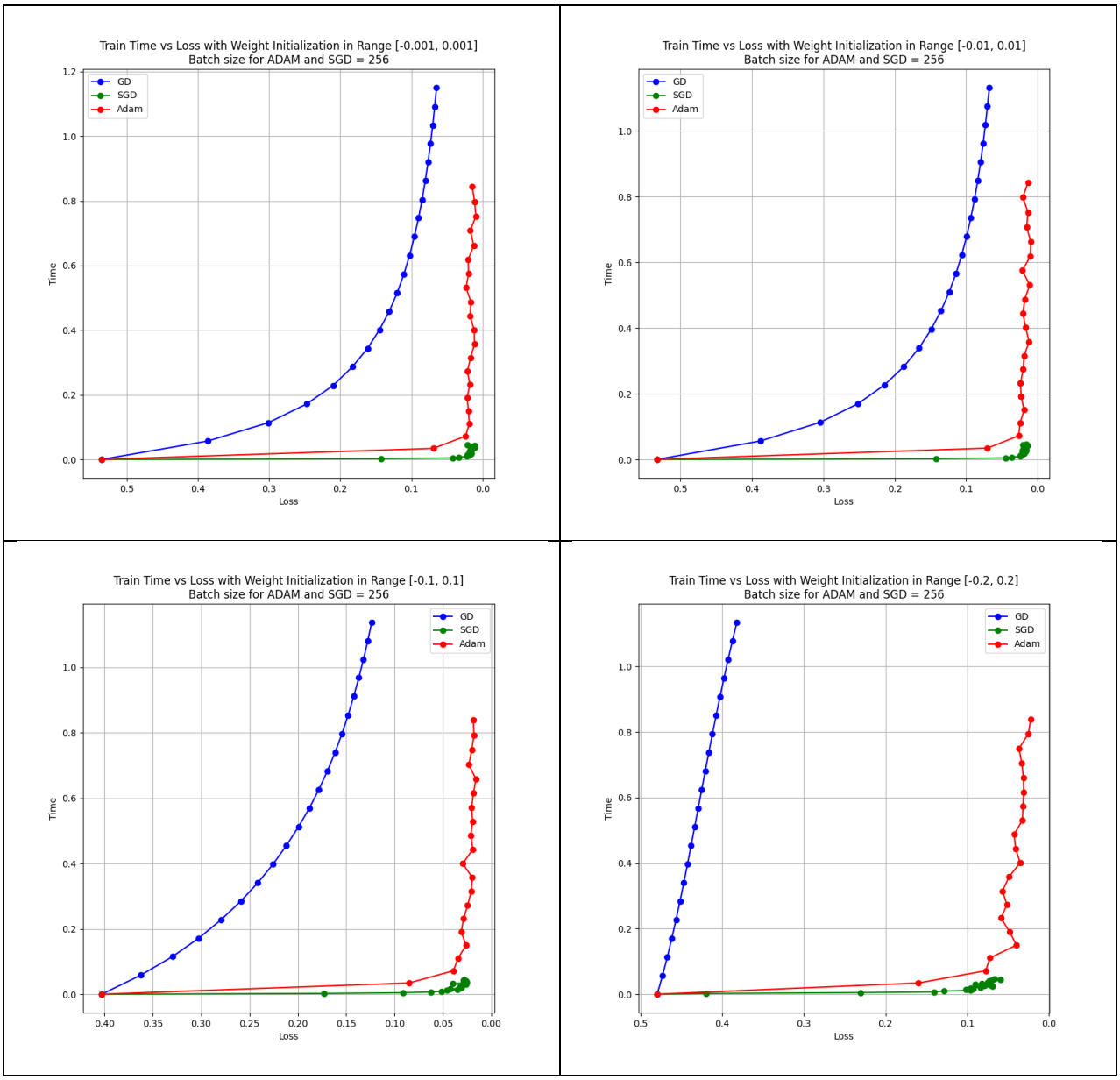
Toplam Doğruluk / Zaman Grafikleri



Test süreleri her üç algoritma için de aynı olduğu için buralardaki yorumumuz diğer grafiklerle aynı olacaktır.

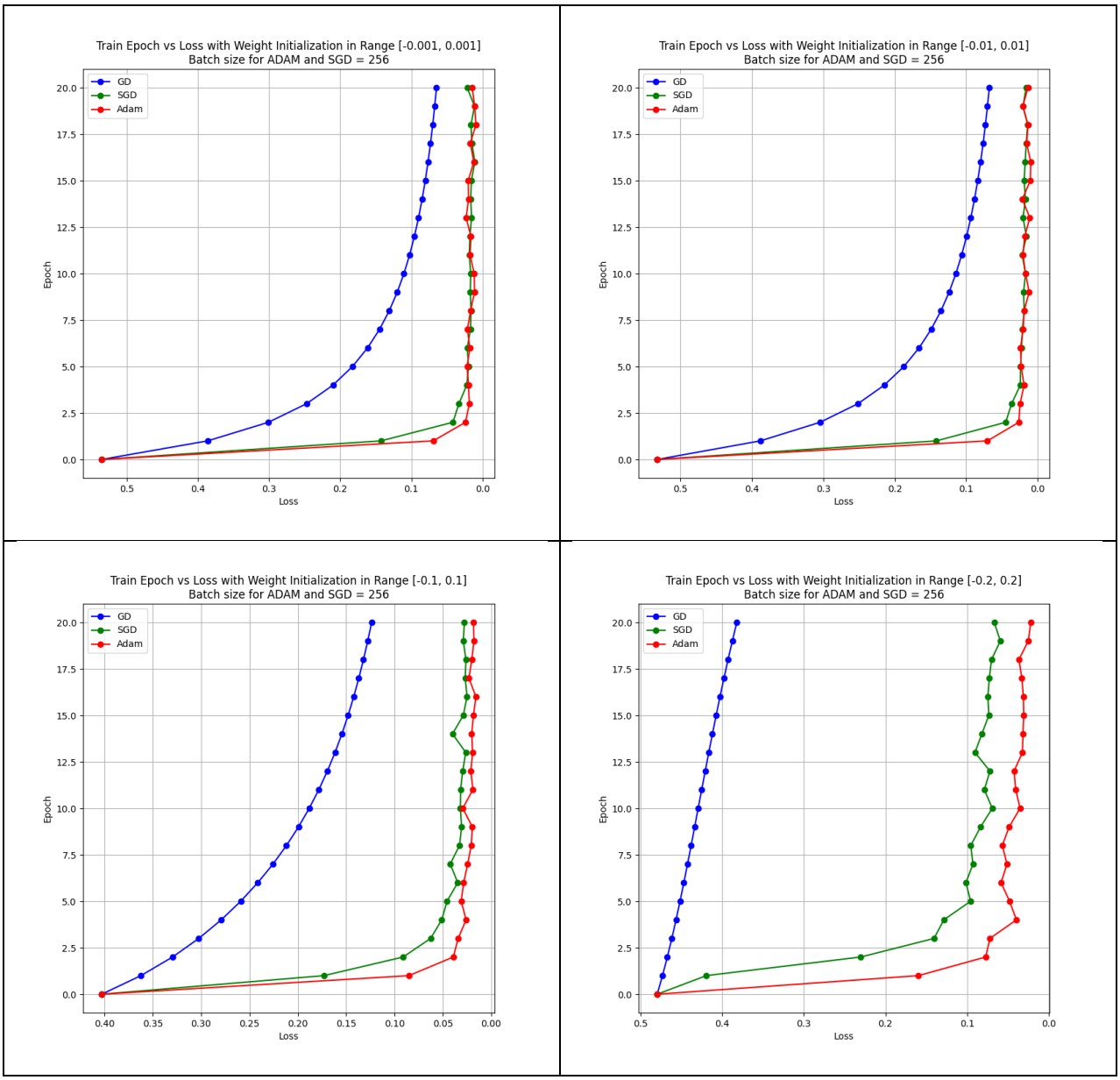
Süre / Kayıp & Güncelleme Sayısı / Kayıp Grafikleri

Eğitim Süre / Kayıp Grafikleri



Sonuçlar ve yorumlar eğitim grafikleriyle aynıdır. Sadece gösterim şekli olarak farklıdır.

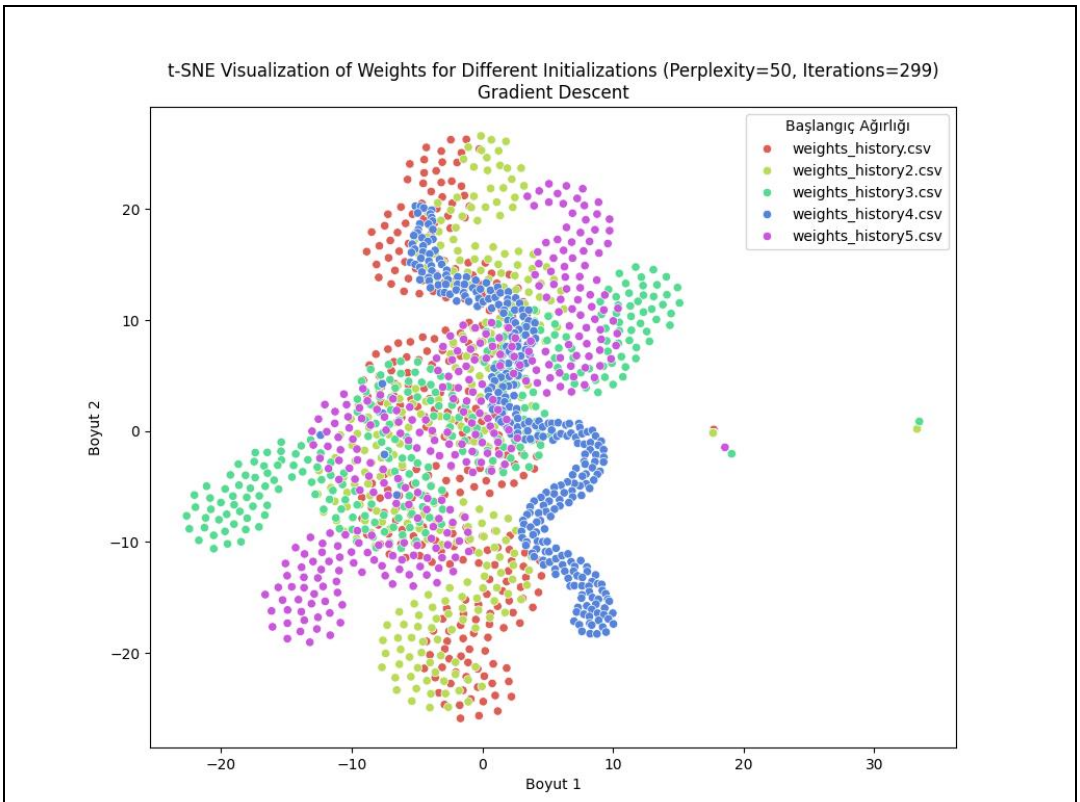
Eğitim Güncelleme Sayısı / Kayıp Grafikleri

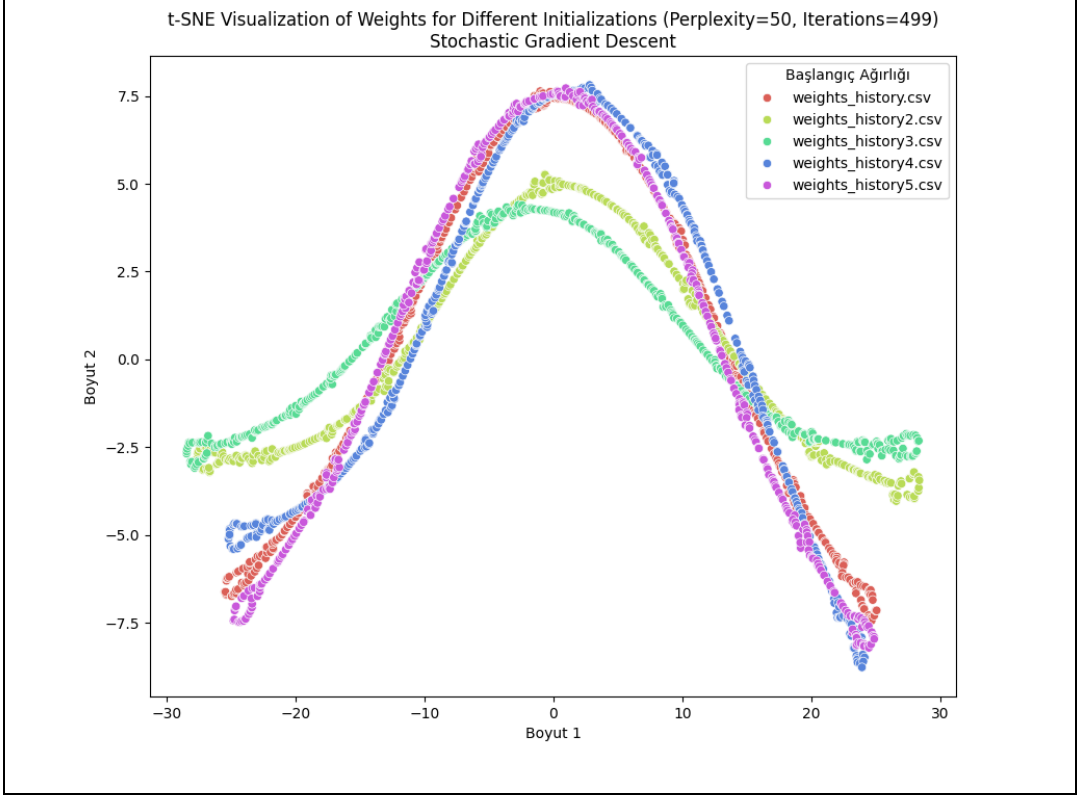


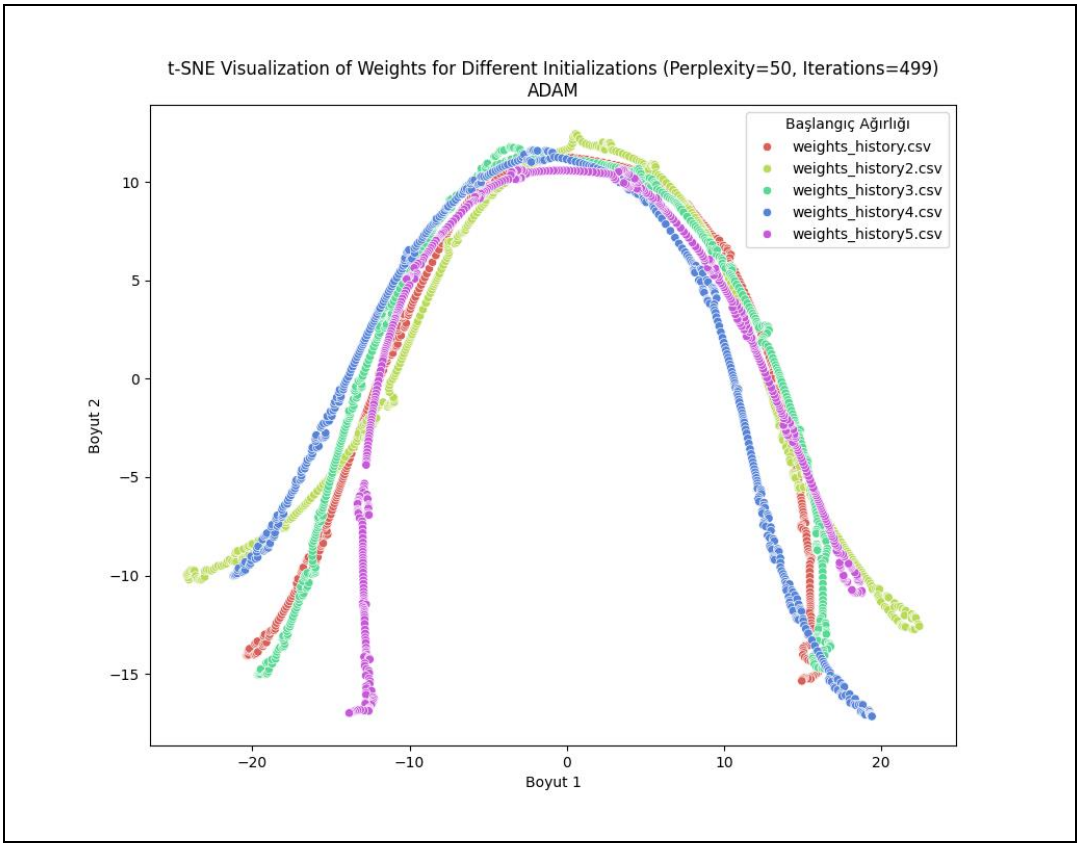
Sonuçlar ve yorumlar eğitim grafikleriyle aynıdır. Sadece gösterim şekli olarak farklıdır.

TSNE Grafikleri

Ağırlıklar random olarak her biri farklı aralıklarla olacak şekilde 5 kere seçildi. {0.001, 0.01, 0.1, 0.3, 0.5}







Grafikler incelendiğinde:

GD, dağınık bir küme yapısı izliyor. Başlangıç noktaları sonuçta çok etkili olmuş. Noktaların çok yayılması ise GD'nin yakınsamak için daha fazla iterasyon sayısına ihtiyaç duyduğunu gösteriyor. Belli noktalara yoğunlaşma ise çok az.

SGD, parabol şekli izliyor. Başlangıç noktaları çok fazla etkilememiş. Fakat büyük ağırlık aralığı değerleri için (0.3 ve 0.5) yakınsama yeterli olmamış. Diğer üç ağırlık aralığı için başlangıç noktaları farklı olmasına rağmen yakınsadıkları nokta aynı gözüküyor. 499 iterasyon yeterli olmuş diyebiliriz.

ADAM, SGD'ye çok benzer bir yapı izlemiş fakat yüksek başlangıç değerlerinden etkilenmemiş gibi duruyor. 499 iterasyon yakınsama için yeterli olmuş.

Sonuç olarak ADAM ve SGD düzgün bir şekilde yakınsarken, GD dağınık bir kümelenme izlemiş. Daha yavaş yakınsanmış ve iterasyonlar yeterli olmamış.
Ağırlıklardan etkilenme sırası şu şekilde olmuş: GD > SGD > ADAM
Bu grafiklere bakarak ADAM ve SGD'nin kullandığımız veri seti için iyi bir performans gösterdiğini söyleyebiliriz. GD ise yetersiz kalmış.

KODLAR VE AÇIKLAMALARI

Bazı fonksiyonlar gerek değişken adları gerek kod blokları olarak anlaşılır olduğundan dolayı kodlara ekstra yorum satırı eklemedim.

Bazı bağımsız fonksiyonlar ve kodlar

Ortak kullanılan sabitler

```
#define IMAGE_SIZE (28 * 28)   // MNIST goruntuleri icin 754 pixel boyutunda
#define MAX_LINE_LENGTH 10000 // csv icin gerekli
#define TRAIN_RATIO 0.8     // Egitim kumesi orani
#define DATASET_SAMPLE_COUNT 43000 // 42000 data var fakat daha fazla yer açmakta fayda var
#define MAX_SAMPLE_COUNT 9000 // anlik islevi yok sonradan ekleneb'l'r

#define BATCH_SIZE 256 // SGD ve ADAM icin ortak batch size boyutu
#define EPOCHS 20
#define LEARNING_RATE_GD 0.01
#define LEARNING_RATE_SGD 0.001
#define LEARNING_RATE_ADAM 0.001
#define WEIGHT_RANGE 0.001

// Adam
#define BETA1 0.9
#define BETA2  0.999
#define EPSILON 1e-8
```

Zaman hesabı kodu hazır olarak alındı. Clock() fonksiyonundan daha hassas olduğu için tercih edildi.

```
// Hazir kod - Zaman olcumu icin gerekli
double get_time() {
    struct timeval tv;
    gettimeofday(&tv, NULL);
    return tv.tv_sec + tv.tv_usec / 1000000.0;
}
```

Pythonda hızlı bir şekilde grafik oluşturabilmek için Array'leri print etmek için kullanılan kod.

```
void print_array(const char* name, float* arr, int size, FILE* file) {
    fprintf(file, "%s = [", name);
    for (int i = 0; i < size; i++) {
        fprintf(file, "%.5f", arr[i]);
        if (i < size - 1) fprintf(file, ", ");
    }
    fprintf(file, "]\n");
}
```

Veri Yükleme ve Normalizasyon

Verilere daha kolay erişmek için MNIST dosyaları yerine Kaggle.com üzerinden .csv olarak veri seti alındı. Veri boyutu MAX_SAMPLE_COUNT sabiti ile sınırlandırılabilir.

```
void read_csv(const char *file_path, int **labels, unsigned char ***images, int
*num_samples) {
    FILE *file = fopen(file_path, "r");
    if (!file) {
        printf("File not found!: %s\n", file_path);
        exit(1);
    }

    char line[MAX_LINE_LENGTH];
    int sample_count = 0;

    *labels = (int *)malloc(sizeof(int) * DATASET_SAMPLE_COUNT);
    *images = (unsigned char **)malloc(sizeof(unsigned char *) *
DATASET_SAMPLE_COUNT);

    fgets(line, sizeof(line), file);

    while (fgets(line, sizeof(line), file)) {
        if (sample_count >= MAX_SAMPLE_COUNT) break;
        char *token;
        int pixel_index = 0;

        token = strtok(line, ",");
        int label = atoi(token);

        if (label == 0 || label == 1) {
            (*labels)[sample_count] = label;
            (*images)[sample_count] = (unsigned char *)malloc(sizeof(unsigned char)
* IMAGE_SIZE);

            while ((token = strtok(NULL, ",")) != NULL) {
                (*images)[sample_count][pixel_index] = (unsigned char)atoi(token);
                pixel_index++;
            }

            sample_count++;
        }
    }

    *num_samples = sample_count;
    fclose(file);
}
```

Pixel değerleri 0-255 arasında olduğu için direkt olarak 255'e bölmek 0-1 aralığına çekmek için yeterli oldu.

```
void normalize_images(unsigned char **images, float ***normalized_images, int num_samples) {
    *normalized_images = (float **)malloc(sizeof(float *) * num_samples);

    for (int i = 0; i < num_samples; i++) {
        (*normalized_images)[i] = (float *)malloc(sizeof(float) * IMAGE_SIZE);
        for (int j = 0; j < IMAGE_SIZE; j++) {
            (*normalized_images)[i][j] = images[i][j] / 255.0f;
        }
    }
}
```

Veri setini bölme fonksiyonu

```
void split(float **images, int *labels, float ***train_images, int **train_labels,
               float ***test_images, int **test_labels, int num_samples) {

    int train_count = (int)(num_samples * TRAIN_RATIO);
    int test_count = num_samples - train_count;

    *train_images = (float **)malloc(sizeof(float *) * train_count);
    *train_labels = (int *)malloc(sizeof(int) * train_count);

    *test_images = (float **)malloc(sizeof(float *) * test_count);
    *test_labels = (int *)malloc(sizeof(int) * test_count);

    for (int i = 0; i < train_count; i++) {
        (*train_images)[i] = images[i];
        (*train_labels)[i] = labels[i];
    }

    for (int i = 0; i < test_count; i++) {
        (*test_images)[i] = images[train_count + i];
        (*test_labels)[i] = labels[train_count + i];
    }
}
```

Ağırlıkların Başlatılması

Ağırlıklar random olarak başlatıldı. Kolay test edilebilirlik ve kod okunulurluğu açısından WEIGHT_RANGE sabiti ile hesap yapıldı.

```
void initialize_weights(float *weights, int input_size) {
    for (int i = 0; i < input_size; i++) {
         weights[i] = (((float)rand() / RAND_MAX) * (WEIGHT_RANGE * 2)) - WEIGHT_RANGE;
    }
}
```

Kayıp Fonksiyonu

Kayıp fonksiyonu olarak MSE kullanıldı.

```
float calculate_mse(float prediction, int target) {
    return (prediction - target) * (prediction - target);
}
```

Aktivasyon Fonksiyonu

Aktivasyon fonksiyonu olarak verdiğiniz ödev dosyasında belirtilen gibi tanh fonksiyonu kullanıldı. Tanh ile ilgili bazı işlemler eğitim fonksiyonlarında direkt olarak uygulanacaktır.

```
float tanh_activation(float x) {
    return tanh(x);
}
```

İleri Yayılım Fonksiyonu

İleri yayılım fonskiyonunda matris çarpımları bias terimi de eklenerek tanh fonksiyonuna gönderilir ve -1,1 aralığında normalize edilerek eğitim fonksiyonlarına döndürülür.

```
float forward(float *weights, float *input, int size) {
    float dot_product = 0.0f;
    for (int i = 0; i < size - 1; i++) {
        dot_product += weights[i] * input[i];
    }
    dot_product += weights[size - 1]; // bias için sakın silme
    return tanh_activation(dot_product);
}
```

Ağırlık Güncelleme Fonksiyonları

Gradient descent için öncelikle **tüm örnekler** için tahmin yapılır. Sonrasında hata hesaplanır. Tüm gradyanlar bulunur ve son olarak gradyanların ortalaması alınarak ağırlıklar güncellenir.

```
void update_weights_gd(float *weights, float **inputs, int *labels, float learning_rate, int
num_samples, int size) {
    float *gradients = (float *)calloc(size, sizeof(float));

    for (int i = 0; i < num_samples; i++) {
        float prediction = forward(weights, inputs[i], size); // tahmin
        float error = prediction - labels[i]; // hata hesabı
        float derivative = 1.0f - (prediction * prediction); // tanh türevi

        for (int j = 0; j < size - 1; j++) {
            gradients[j] += error * derivative * inputs[i][j];
        }
        gradients[size - 1] += error * derivative; // bias icin gradyan
    }

    for (int i = 0; i < size; i++) {
        weights[i] -= learning_rate * gradients[i] / num_samples;
    }

    free(gradients);
}
```

Stochastic Gradient Descent için yanlızca bir veri örneğinin ağırlıkları güncellenir. BATCH_SIZE ile döngü bu fonksiyonda değil eğitim fonksiyonunda yapılır.

```
void update_weights_sgd(float *weights, float *input, int label, float learning_rate, int
size) {
    float prediction = forward(weights, input, size);
    float error = prediction - label;

    for (int i = 0; i < size - 1; i++) {
        weights[i] -= learning_rate * error * (1.0f - prediction * prediction) * input[i];
    }
    weights[size - 1] -= learning_rate * error * (1.0f - prediction * prediction);
}
```

ADAM için de önce gradyanları hesaplarız. Sonrasında ADAM'a özgü sabitleri kullanarak momentum hesabı yapariz. Son aşamada ağırlıkları öğrenim oranını (learning rate) yenileyerek güncelleriz. Bu şekilde adaptif öğrenme oranını uygulamış oluruz.

```
void update_weights_adam(float *weights, int *labels, int index, float **images, int size,
float *first_moment, float *second_moment , int iter, float learning_rate) {

 // Gradyan hesabi
    float prediction = forward(weights, images[index], size);
    float *gradients = (float *)calloc(size, sizeof(float));
    float error = prediction - labels[index];
    float derivative = 1.0f - (prediction * prediction); //tanh türev

    for (int j = 0; j < size - 1; j++) {
        gradients[j] = error * derivative * images[index][j];
    }
    gradients[size - 1] = error * derivative;


    for (int i = 0; i < size; i++) {

        first_moment [i] = BETA1 * first_moment [i] + (1 - BETA1) * gradients[i];
        second_moment [i] = BETA2 * second_moment [i] + (1 - BETA2) * gradients[i] *
gradients[i];


        float corrected_first_moment = first_moment [i] / (1 - powf(BETA1, iter));
        float corrected_second_moment  = second_moment[i] / (1 - powf(BETA2, iter));

        weights[i] -= learning_rate * corrected_first_moment / (sqrtf(corrected_second_moment
) + EPSILON);

    }

    free(gradients);

}
```

Eğitim Fonksiyonları

Gradient Descent için her epochta tüm örnekler üzerinden ağırlık güncellemesi yapılır. Öncelikle ileri yayılım yapılarak tahmin oluşturulur, sonrasında tahmin değerlendirilir ve ağırlıklar güncellenir.

```
void train_gd(float **images, int *labels, int num_samples, float *weights, int size) {
    float final_accuracy = 0;
    float final_loss = 0;

    printf("Training with Gradient Descent...\n");
    for (int epoch = 0; epoch < EPOCHS; epoch++) {
        double epoch_start_time = get_time();
        float total_loss = 0;
        int correct_predictions = 0;

        // forward
        for (int i = 0; i < num_samples; i++) {
            float prediction = forward(weights, images[i], size);
            total_loss += calculate_mse(prediction, labels[i]);
            if ((prediction >= 0.5 && labels[i] == 1) || (prediction < 0.5 && labels[i] == 0))
{
                correct_predictions++;
            }
        }

        // backward
        update_weights_gd(weights, images, labels, LEARNING_RATE_GD, num_samples, size);

        // ÇIKTI

        float avg_loss = total_loss / num_samples;
        float accuracy = (float)correct_predictions / num_samples * 100;
        final_accuracy = accuracy;
        final_loss = avg_loss;

        double epoch_end_time = get_time();

        // Python için çıktı
        train_loss_gd[epoch + 1] = avg_loss;
        train_accuracy_gd[epoch + 1] = accuracy;
        train_time_gd[epoch + 1] = train_time_gd[epoch] + epoch_end_time - epoch_start_time;


        printf("Epoch %d | Loss: %.5f | Accuracy: %.3f%% | Time: %.4f seconds\n",
               epoch + 1, avg_loss, accuracy, epoch_end_time - epoch_start_time);

    }
    printf("\nGradient Descent Training Accuracy: %.3f%%\n", final_accuracy);
    printf("Gradient Descent Final Training Loss: %.5f\n", final_loss);

}
```

Stochastic Gradient Descent için GD'deki aynı adımlar uygulanır. Tek fark her epochta tüm örnekler yerine BATCH_SIZE sabiti kadar (default olarak 256) güncelleme yapılır.

```
void train_sgd(float **images, int *labels, int num_samples, float *weights, int size) {
    float final_accuracy = 0;
    float final_loss = 0;


    printf("Training with Stochastic Gradient Descent...\n");

    for (int epoch = 0; epoch < EPOCHS; epoch++) {
        double epoch_start_time = get_time();
        float total_loss = 0.0f;
        int correct_predictions = 0;


        for (int step = 0; step < BATCH_SIZE; step++) {
            int index = rand()%num_samples;

           // Forward
            float prediction = forward(weights, images[index], size);
            total_loss += calculate_mse(prediction, labels[index]);

            int prediction_class = (prediction >= 0.5f) ? 1 : 0;
            if (prediction_class == labels[index]) {
                correct_predictions++;
            }

             // Backward
            update_weights_sgd(weights, images[index], labels[index], LEARNING_RATE_SGD,
size);
        }


        // ÇIKTI

        float avg_loss = total_loss / BATCH_SIZE;
        float accuracy = (float)correct_predictions / BATCH_SIZE * 100.0f;
        final_accuracy = accuracy;
        final_loss = avg_loss;

        double epoch_end_time = get_time();

        // Python için çıktı
        train_loss_sgd[epoch + 1] = avg_loss;
        train_accuracy_sgd[epoch + 1] = accuracy;
        train_time_sgd[epoch + 1] = train_time_sgd[epoch] + epoch_end_time - epoch_start_time;
        printf("Epoch %d | Loss: %.5f | Accuracy: %.3f%% | Time: %.4f seconds\n",
               epoch + 1, avg_loss, accuracy, epoch_end_time - epoch_start_time);

    }
    printf("\nStochastic Gradient Descent Training Accuracy: %.3f%%\n", final_accuracy);
    printf("Stochastic Gradient Descent Final Training Loss: %.5f\n", final_loss);

}
```

Adam için SGD'deki aynı adımlar uygulanır fakat momentum hesabındaki formül için iterasyon sayısının tutulması ve update_weights_adam fonksiyonuna gönderilmesi gereklidir.

```
void train_adam(float **images, int *labels, int num_samples, float *weights, int size) {
    float *first_moment = (float *)calloc(size, sizeof(float));
    float *second_moment = (float *)calloc(size, sizeof(float));
    int iteration = 0;

    float final_accuracy = 0;
    float final_loss = 0;

    printf("Training with ADAM...\n");

    for (int epoch = 0; epoch < EPOCHS; epoch++) {
        double epoch_start_time = get_time();
        float total_loss = 0.0f;
        int correct_predictions = 0;

        for (int step = 0; step < BATCH_SIZE; step++) {
            int index = rand()%num_samples;
            // Forward
            float prediction = forward(weights, images[index], size);
            total_loss += calculate_mse(prediction, labels[index]);

             prediction = (prediction >= 0.5) ? 1 : 0;
            if (prediction == labels[index]) {
                correct_predictions++;
            }

            // Backward
            iteration++;
            update_weights_adam(weights, labels, index, images, size, first_moment,
second_moment, iteration, LEARNING_RATE_ADAM);

        }

        float avg_loss = total_loss / BATCH_SIZE;
        float accuracy = (float)correct_predictions / BATCH_SIZE * 100.0f;
        final_accuracy = accuracy;
        final_loss = avg_loss;

        double epoch_end_time = get_time();

        // Python için çıktı
        train_loss_adam[epoch + 1] = avg_loss;
        train_accuracy_adam[epoch + 1] = accuracy;
        train_time_adam[epoch + 1] = train_time_adam[epoch] + epoch_end_time -
epoch_start_time;

        printf("Epoch %d | Loss: %.5f | Accuracy: %.3f%% | Time: %.4f seconds\n",
               epoch + 1, avg_loss, accuracy, epoch_end_time - epoch_start_time);

    }

    printf("\nADAM Training Accuracy: %.3f%%\n", final_accuracy);
    printf("ADAM Final Training Loss: %.5f\n", final_loss);

    free(first_moment);
    free(second_moment);
}
```

Main fonksiyonu

Test Kısmı (Okuma kolaylığı açısından main fonksiyonundan ayrı yazıldı)

```
// Test Asamasi
int test_count = num_samples - (int)(num_samples * TRAIN_RATIO);
float test_loss_gd[test_count / 100 + 1], test_accuracy_gd[test_count / 100 + 1];
float test_loss_sgd[test_count / 100 + 1], test_accuracy_sgd[test_count / 100 + 1];
float test_loss_adam[test_count / 100 + 1], test_accuracy_adam[test_count / 100 + 1];

float test_times[test_count / 100 + 1];

float *test_weights[] = {weights_gd, weights_sgd, weights_adam};
float *test_loss[] = {test_loss_gd, test_loss_sgd, test_loss_adam};
float *test_accuracy[] = {test_accuracy_gd, test_accuracy_sgd, test_accuracy_adam};
char *algorithm_names[] = {"GD", "SGD", "ADAM"};

for (int alg = 0; alg < 3; alg++) {
    float total_test_time = 0;
    int test_output_count = 0;
    int correct_predictions = 0;
    float total_loss = 0.0f;

    for (int i = 0; i < test_count; i++) {

        start_time = get_time();
        float prediction = forward(test_weights[alg], test_images[i], input_size);
        prediction = (prediction >= 0.5) ? 1 : 0;

        if (prediction == test_labels[i]) {
            correct_predictions++;
        }

        total_loss += calculate_mse(prediction, test_labels[i]);

        end_time = get_time();
        total_test_time += end_time - start_time;
        if (i % 100 == 0) {
            float accuracy = (float)correct_predictions / (i + 1) * 100.0f;
            float loss = (float)total_loss / (i + 1);

            test_loss[alg][test_output_count] = loss;

            test_accuracy[alg][test_output_count] = accuracy;

            test_times[test_output_count] = total_test_time;

            test_output_count++;
            printf("%s TEST, Step: %d, Total Loss: %.5f, Total Accuracy: %.2f%%\n",
algorithm_names[alg],i, loss, accuracy);
        }
    }

    float accuracy = (float)correct_predictions / test_count * 100.0f;
    float avg_loss = total_loss / test_count;

    printf("%s Test Accuracy: %.2f%%\n", algorithm_names[alg], accuracy);
    printf("%s Test Loss: %.5f\n\n", algorithm_names[alg], avg_loss);
}
```

Main fonksiyonunun test kısmı çıkarılmış hali:

```c
int main() {
    srand(time(NULL));

    const char *file_path = "data/train.csv";
    int *labels = NULL;
    unsigned char **images = NULL;
    int num_samples = 0;

    read_csv(file_path, &labels, &images, &num_samples);

    float **normalized_images = NULL;
    normalize_images(images, &normalized_images, num_samples);

    float **train_images = NULL, **test_images = NULL;
    int *train_labels = NULL, *test_labels = NULL;
    split(normalized_images, labels, &train_images, &train_labels, &test_images, &test_labels,
num_samples);

    int input_size = IMAGE_SIZE + 1;

    // Ağırlıkları bir kez initialize et
    float *weights = (float *)malloc(sizeof(float) * input_size);
    initialize_weights(weights, input_size);

    // GD, SGD ve Adam için ağırlık kopyaları
    float *weights_gd = (float *)malloc(sizeof(float) * input_size);
    memcpy(weights_gd, weights, sizeof(float) * input_size);

    float *weights_sgd = (float *)malloc(sizeof(float) * input_size);
    memcpy(weights_sgd, weights, sizeof(float) * input_size);

    float *weights_adam = (float *)malloc(sizeof(float) * input_size);
    memcpy(weights_adam, weights, sizeof(float) * input_size);

    double start_time, end_time;

    // Başlangıç loss değerlerini hesapla ve kaydet
    float initial_loss = 0.0f;
    for (int i = 0; i < (int)(num_samples * TRAIN_RATIO); i++) {
        float prediction = forward(weights, train_images[i], input_size);
        initial_loss += calculate_mse(prediction, train_labels[i]);
    }

    initial_loss /= (int)(num_samples * TRAIN_RATIO);
    printf("Initial loss: %f\n\n", initial_loss);

    // GD Egitimi
    start_time = get_time();
    train_gd(train_images, train_labels, (int)(num_samples * TRAIN_RATIO), weights_gd, input_size);
    end_time = get_time();
    printf("Gradient Descent total time: %.4f seconds\n\n", end_time - start_time);

    // SGD Egitimi
    start_time = get_time();
    train_sgd(train_images, train_labels, (int)(num_samples * TRAIN_RATIO), weights_sgd, input_size);
    end_time = get_time();
    printf("Stochastic Gradient Descent total time: %.4f seconds\n\n", end_time - start_time);

    // Adam Egitimi
    start_time = get_time();
    train_adam(train_images, train_labels, (int)(num_samples * TRAIN_RATIO), weights_adam, input_size);
    end_time = get_time();
    printf("ADAM total time: %.4f seconds\n\n", end_time - start_time);


    // Test Asamasi

// TEST KODLARI BURAYA GELECEK

////////////////////////////////////////////////////////////////////////////////////////////////////
    int test_count = num_samples - (int)(num_samples * TRAIN_RATIO);
    float test_loss_gd[test_count / 100 + 1], test_accuracy_gd[test_count / 100 + 1];
    float test_loss_sgd[test_count / 100 + 1], test_accuracy_sgd[test_count / 100 + 1];
    float test_loss_adam[test_count / 100 + 1], test_accuracy_adam[test_count / 100 + 1];
```

```
    // Grafik için kolay çıktı alım kısmı

    train_loss_gd[0] = initial_loss;
    train_loss_sgd[0] = initial_loss;
    train_loss_adam[0] = initial_loss;

    FILE* file = fopen("results.txt", "w");
    if (file == NULL) {
        printf("File not found!\n");
        return 0;
    }

    fprintf(file, "epochs = [");
    for (int i = 0; i < EPOCHS + 1; i++) {
        fprintf(file, "%d", i);
        if (i < EPOCHS) fprintf(file, ", ");
    }
    fprintf(file, "]\n");

    fprintf(file, "test_steps = [");
    for (int i = 0; i < test_count / 100 + 1; i++) {
        fprintf(file, "%d", i * 100);
        if (i < test_count / 100) fprintf(file, ", ");
    }
    fprintf(file, "]\n");

    // Algoritmaların loss ve accuracy değerlerini yazdırma
    print_array("train_loss_gd", train_loss_gd, EPOCHS + 1, file);
    print_array("train_loss_sgd", train_loss_sgd, EPOCHS + 1, file);
    print_array("train_loss_adam", train_loss_adam, EPOCHS + 1, file);


    print_array("train_accuracy_gd", train_accuracy_gd, EPOCHS + 1, file);
    print_array("train_accuracy_sgd", train_accuracy_sgd, EPOCHS + 1, file);
    print_array("train_accuracy_adam", train_accuracy_adam, EPOCHS + 1, file);


    print_array("train_time_gd", train_time_gd, EPOCHS + 1, file);
    print_array("train_time_sgd", train_time_sgd, EPOCHS + 1, file);
    print_array("train_time_adam", train_time_adam, EPOCHS + 1, file);

    print_array("test_loss_gd", test_loss_gd, test_count / 100 + 1, file);
    print_array("test_loss_sgd", test_loss_sgd, test_count / 100 + 1, file);
    print_array("test_loss_adam", test_loss_adam, test_count / 100 + 1, file);

    print_array("test_accuracy_gd", test_accuracy_gd, test_count / 100 + 1, file);
    print_array("test_accuracy_sgd", test_accuracy_sgd, test_count / 100 + 1, file);
    print_array("test_accuracy_adam", test_accuracy_adam, test_count / 100 + 1, file);

    print_array("test_times", test_times, test_count / 100 + 1, file);

    fprintf(file, "weight_range = %0.3f\n", WEIGHT_RANGE);
    fprintf(file, "batch_size = %d", BATCH_SIZE);

    // Dosyayı kapatma
    fclose(file);

    printf("Results are saved to 'results.txt' file.\n");

    free(labels);
    for (int i = 0; i < num_samples; i++) {
        free(images[i]);
        free(normalized_images[i]);
    }

    free(images);
    free(normalized_images);
    free(train_labels);
    free(test_labels);
    free(train_images);
    free(test_images);
    free(weights);
    free(weights_gd);
    free(weights_sgd);
    free(weights_adam);

    return 0;
}
```

TARTIŞMA

Elde edilen sonuçlar şu şekilde yorumlanabilir:

Algoritma Performansı:

GD, hem doğruluk bakımından hem de hız bakımından her iki algoritmanın da gerisinde kaldı. Tüm örnekleri işleme aldığı için büyük veri setlerine kesinlikle uygun değil.

SGD, her iterasyonda yalnızca bir parti boyutunda işlem yaptığı için daha gürültülü bir öğrenme süreci geçirse de ADAM'a çok yakın sonuçlar aldı. Hız olarak en hızlısıydı. Bu onu büyük veri setleri için daha uygun kılar.

ADAM, diğer algoritmalara kıyasla daha stabil bir şekilde eğitim sürecini tamamladı ve daha iyi sonuçlar elde etti. Süre olarak SGD'den yavaş, GD'den ise çok daha hızlıydı. Bu, ADAM'ın, her parametre için farklı öğrenme oranları kullanması ve momentumu dikkate alarak daha etkili bir şekilde öğrenme yapmasıyla açıklanabilir. Aynı parti boyutuna sahip olmasına rağmen SGD'den daha yavaş çalışmasının sebebi de daha fazla işlem yapmasıdır.

Daha hassas modeller için ADAM, daha büyük veri setleri için SGD daha iyi olabilir.

Batch Boyutu ve Öğrenme Oranı:

Eğitimde kullanılan batch size değeri, algoritmaların hızını ve doğruluğunu doğrudan etkilemiştir. Küçük batch boyutları genellikle daha hızlı olsa da doğruluk değerlerini düşürebilir. Diğer taraftan, büyük batch size değerleri daha yüksek doğruluğa sahip olsa da zaman bakımından verimsiz olması sebebiyle SGD ve ADAM'ı GD'nin hızına yaklaştırabilir. Grafikleri incelediğimizde kullandığımız veri seti için en verimli parti boyutunun 128 ile 256 arasında olduğu görülmekte.

Öğrenme oranı da algoritmaların başarısını etkileyen önemli bir faktördür. Yüksek oranlar hızlı bir öğrenme sunarken dengesiz bir yol izler. Yerel minimumlara takılma ihtimali artar. Düşük öğrenme oranları ise daha dengeli bir süreç sunarken öğrenim süresini arttıracaktır. Örneğin veri setimiz için ADAM, 0.01 öğrenme oranında %52 doğruluk vermektedir.

Kullandığımız veri seti için yapılan testlere göre aşağıdaki öğrenim oranları en optimal sonucu vermiştir:

GD öğrenme oranı = 0.01

SGD ve ADAM'ın öğrenme oranı = 0.001

Modelin Sınırlamaları:

Bu projede kullanılan basit yapılar ve optimizasyon algoritmaları, özellikle derin ağlar gibi daha karmaşık yapıları öğrenmek için yetersiz kalabilir. Örneğin, gizli katman kullanılmaması, modelin daha karmaşık ilişkileri öğrenme kapasitesini sınırlamıştır.

MNIST veri seti gibi kolay bir veri seti kullanıldığından daha karmaşık veri setlerinde sonuçlar istenen gibi olmayabilir.

Sonuçların Genellenebilirliği:

Sonuçlar, yalnızca MNIST veri seti üzerindeki ikili sınıflandırma için geçerli olabilir. Fakat genel olarak GD, SGD ve ADAM algoritmalarının sonuçları diğer araştırmalarda bulunan sonuçlara çok yakındır.

SONUÇ

MNIST data setinde yapılan ikili sınıflandırma için GD, SGD ve ADAM algoritmalarının karşılaştırması aşağıdaki gibidir:

Stabilite:
GD > ADAM > SGD

- GD çok daha stabil sonuçlar verdi. ADAM ve SGD algoritmalarında parti boyutu kullanılması sebebiyle dalgalanmalar meydana geldi. ADAM algoritması kullandığı yöntemler sayesinde SGD'den daha stabil sonuçlar elde etti.

Doğruluk:
ADAM > SGD > GD

- ADAM ve SGD çok yakın doğruluk değerleri alsa da ADAM ortalama olarak daha fazla doğruluğa sahipti. GD ise iki algoritmanın da çok gerisinde kaldı.

Hız:
SGD > ADAM > GD

- Parti boyutuyla değişmekle beraber SGD, GD algoritmasından onlarca hatta yüzlerce kez daha hızlıydı. ADAM ise yaptığı işlemler sebebiyle SGD'nin gerisinde kaldı.

Sonuç olarak MNIST data setinde yapılan ikili sınıflandırma için SGD algoritması yüksek hızı ve doğruluğu sebebiyle diğer iki algoritmadan daha verimli olabilir. Büyük veri setleri için SGD daha verimli olacaktır.
ADAM algoritması ise dengeli bir öğrenme sürecine sahip olması ve en yüksek doğruluk değerini vermesi sebebiyle, iki algoritmanın ortasında bir hıza sahip olduğunu hesaba

katarsak daha hassas veri setlerinde ve orta boyutlu veri setlerinde daha iyi bir yöntem olabilir.
GD ise üç algoritma arasından en dengelisi olması ve doğruluk değeri bakımından çok fazla geride kalmaması sebebiyle, en yavaş öğrenim hızına sahip algoritma olduğunu da ele alırsak küçük veri setlerinde ve yüksek güncelleme sayısına sahip modellerde daha iyi sonuçlar verebilir.

KAYNAKÇA

Kullanılan veri seti: *https://www.kaggle.com/datasets/bhavikjikadara/handwritten-digit-recognition*

EKLER

Grafik Çizme Kodları

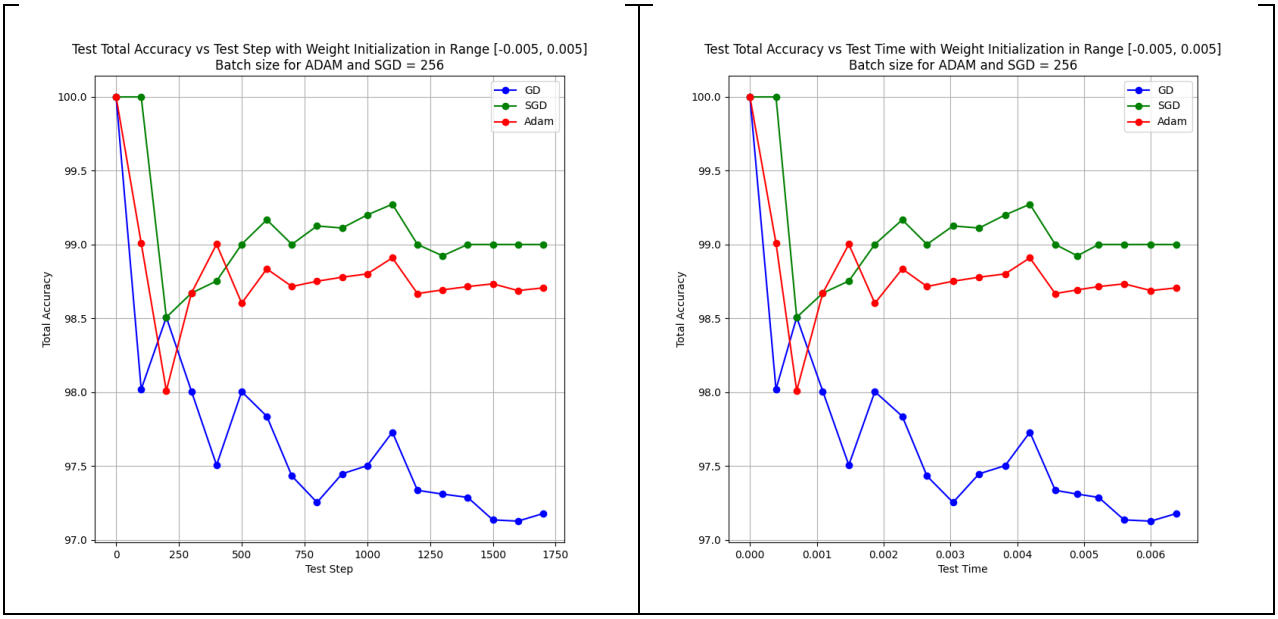
Tüm kodlar github linkinde bulunmaktadır. Örnek bir kod aşağıdaki gibidir:

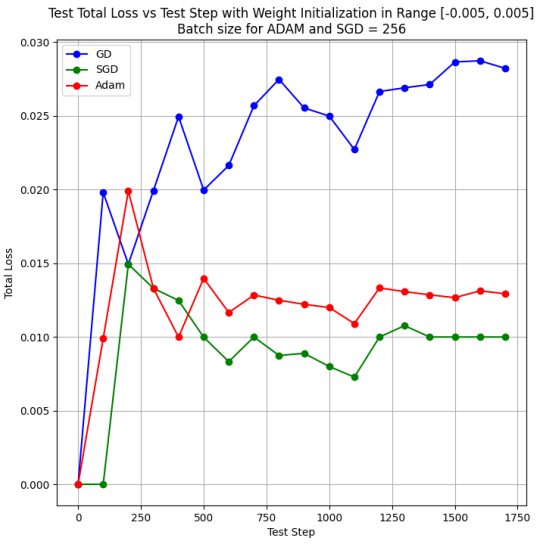
```
import os
import matplotlib.pyplot as plt

# Veriler
epochs = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]
test_steps = [0, 100, 200, 300, 400, 500, 600, 700, 800, 900, 1000, 1100, 1200, 1300, 1400, 1500,
1600, 1700]
train_loss_gd = [0.53157, 0.38534, 0.30085, 0.24744, 0.21043, 0.18301, 0.16181, 0.14494, 0.13124,
0.11996, 0.11055, 0.10261, 0.09583, 0.09000, 0.08494, 0.08051, 0.07660, 0.07313, 0.07003, 0.06725,
0.06473]
train_loss_sgd = [0.53157, 0.14355, 0.04299, 0.03768, 0.02524, 0.03077, 0.01966, 0.01922, 0.01874,
0.02096, 0.01849, 0.01612, 0.01586, 0.01547, 0.01512, 0.01829, 0.02387, 0.01649, 0.01276, 0.01291,
0.01909]
train_loss_adam = [0.53157, 0.06521, 0.03221, 0.02141, 0.02008, 0.01601, 0.01945, 0.02237, 0.01718,
0.01812, 0.01448, 0.01375, 0.01608, 0.02037, 0.01244, 0.01303, 0.01533, 0.00751, 0.01263, 0.02398,
0.01519]
train_accuracy_gd = [0.00000, 46.73851, 46.73851, 47.27737, 53.11968, 62.57799, 70.68916, 77.77935,
83.09699, 86.88316, 89.50652, 91.64776, 92.89563, 94.15768, 94.89507, 95.61826, 96.08622, 96.49744,
96.80942, 96.97958, 97.26319]
train_accuracy_sgd = [0.00000, 84.37500, 97.26563, 98.04688, 99.60938, 98.04688, 99.21875, 98.82813,
99.60938, 99.21875, 99.21875, 99.60938, 99.21875, 99.60938, 99.60938, 98.82813, 97.26563, 99.60938,
99.60938, 100.00000, 98.43750]
train_accuracy_adam = [0.00000, 94.92188, 99.21875, 99.60938, 98.82813, 99.60938, 99.21875,
99.21875, 99.60938, 99.60938, 100.00000, 99.21875, 99.21875, 98.43750, 99.60938, 99.60938, 98.82813,
100.00000, 100.00000, 98.43750, 100.00000]
train_time_gd = [0.00000, 0.08976, 0.16638, 0.25591, 0.33980, 0.43317, 0.52230, 0.61234, 0.70425,
0.78535, 0.88330, 0.97178, 1.04796, 1.14334, 1.23506, 1.31002, 1.40080, 1.49359, 1.58649, 1.67295,
1.75618]
train_time_sgd = [0.00000, 0.01186, 0.01698, 0.01981, 0.02385, 0.02563, 0.03051, 0.03441, 0.03742,
0.04170, 0.04438, 0.04776, 0.05093, 0.05406, 0.05794, 0.05924, 0.06501, 0.06908, 0.07211, 0.07614,
0.07815]
train_time_adam = [0.00000, 0.04162, 0.08533, 0.12690, 0.16912, 0.20858, 0.25375, 0.29767, 0.34130,
0.38389, 0.42696, 0.46987, 0.51126, 0.55578, 0.59839, 0.64157, 0.68440, 0.72700, 0.77011, 0.81155,
0.85390]
test_loss_gd = [0.00000, 0.01980, 0.01990, 0.01993, 0.02494, 0.01996, 0.02163, 0.02568, 0.02747,
0.02442, 0.02298, 0.02089, 0.02498, 0.02460, 0.02498, 0.02665, 0.02623, 0.02587]
test_loss_sgd = [0.00000, 0.00000, 0.00498, 0.00997, 0.00998, 0.00798, 0.00666, 0.00856, 0.00749,
0.00777, 0.00699, 0.00636, 0.00916, 0.00922, 0.00857, 0.00799, 0.00812, 0.00823]
test_loss_adam = [0.00000, 0.00990, 0.01990, 0.01661, 0.01496, 0.01796, 0.01664, 0.01712, 0.01623,
0.01554, 0.01499, 0.01362, 0.01749, 0.01768, 0.01784, 0.01666, 0.01749, 0.01705]
test_accuracy_gd = [100.00000, 98.01980, 98.00995, 98.00665, 97.50623, 98.00399, 97.83694, 97.43224,
97.25343, 97.55827, 97.70230, 97.91099, 97.50208, 97.54035, 97.50179, 97.33511, 97.37664, 97.41328]
test_accuracy_sgd = [100.00000, 100.00000, 99.50249, 99.00332, 99.00249, 99.20160, 99.33444,
```

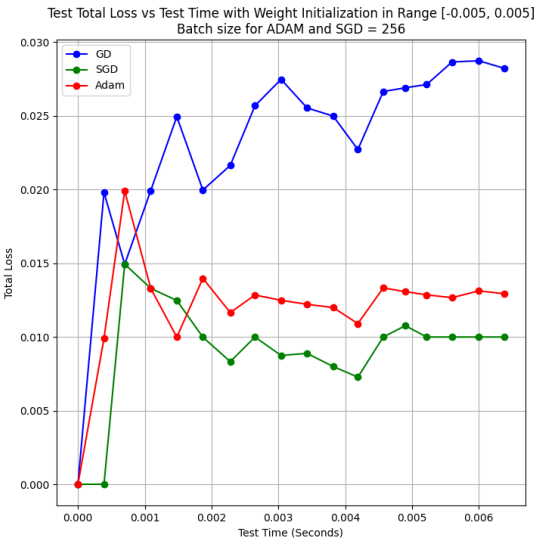
```
99.14408, 99.25094, 99.22308, 99.30070, 99.36421, 99.08410, 99.07763, 99.14347, 99.20053, 99.18801,
99.17696]
test_accuracy_adam = [100.00000, 99.00990, 98.00995, 98.33887, 98.50374, 98.20359, 98.33611,
98.28816, 98.37703, 98.44617, 98.50150, 98.63760, 98.25146, 98.23213, 98.21556, 98.33444, 98.25109,
98.29512]
test_times = [0.00000, 0.00000, 0.00000, 0.00460, 0.00460, 0.00609, 0.00609, 0.00757, 0.00757,
0.00757, 0.00757, 0.01008, 0.01008, 0.01173, 0.01173, 0.01173, 0.01173, 0.01453]
weight_range = 0.001
batch_size = 256

output_folder = "test_loss_step"

os.makedirs(output_folder, exist_ok=True)

plt.figure(figsize=(8, 8))

plt.plot(test_steps, test_loss_gd, label='GD', color='blue', marker='o')
plt.plot(test_steps, test_loss_sgd, label='SGD', color='green', marker='o')
plt.plot(test_steps, test_loss_adam, label='Adam', color='red', marker='o')

plt.title(f'Test Total Loss vs Test Step with Weight Initialization in Range [-{weight_range},
{weight_range}]\nBatch size for ADAM and SGD = {batch_size}')
plt.xlabel('Test Step')
plt.ylabel('Total Loss')

plt.legend()

plt.grid(True)

file_path = os.path.join(output_folder, f'total_loss_vs_test_step_weight_range_{weight_range}.png')

plt.savefig(file_path)

plt.show()
```

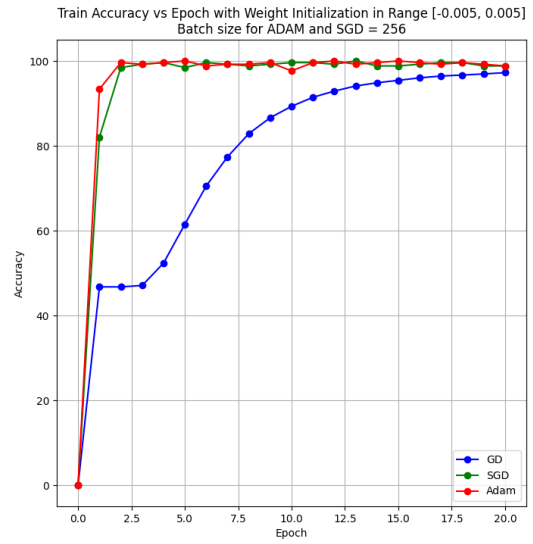
Ekstra Grafikler

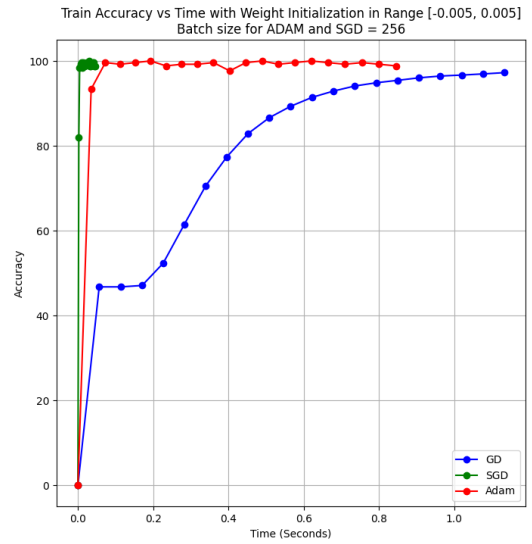
0.005 ağırlık sınırları için grafikler aşağıdaki gibidir:

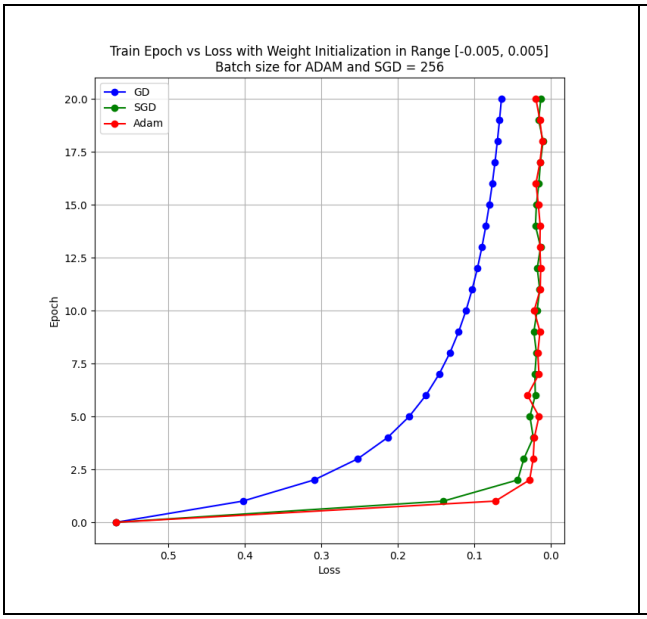


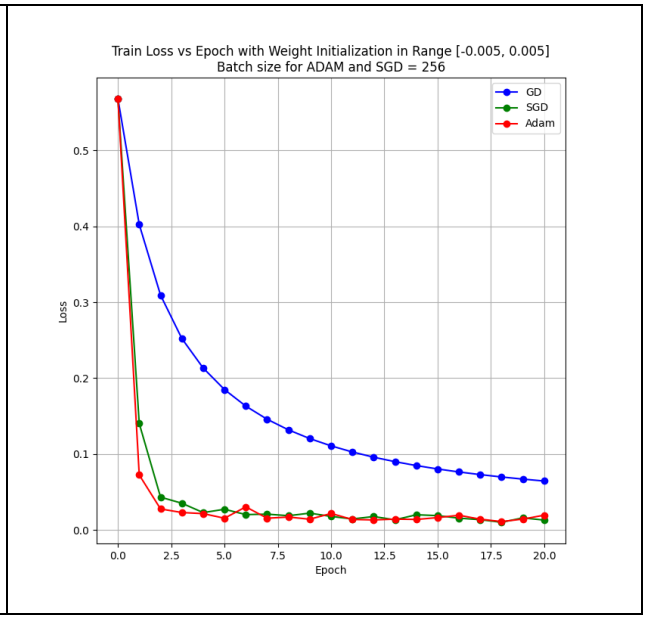

Test Total Loss vs Test Step with Weight Initialization in Range [-0.005, 0.005]
Batch size for ADAM and SGD = 256
GD
SGD
Adam
Total Loss
Test Step

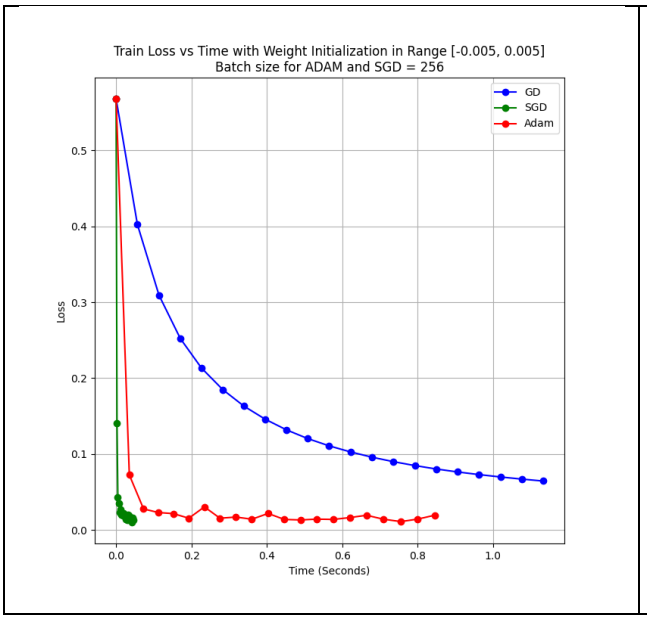

Test Total Loss vs Test Time with Weight Initialization in Range [-0.005, 0.005]
Batch size for ADAM and SGD = 256
GD
SGD
Adam
Total Loss
Test Time (Seconds)

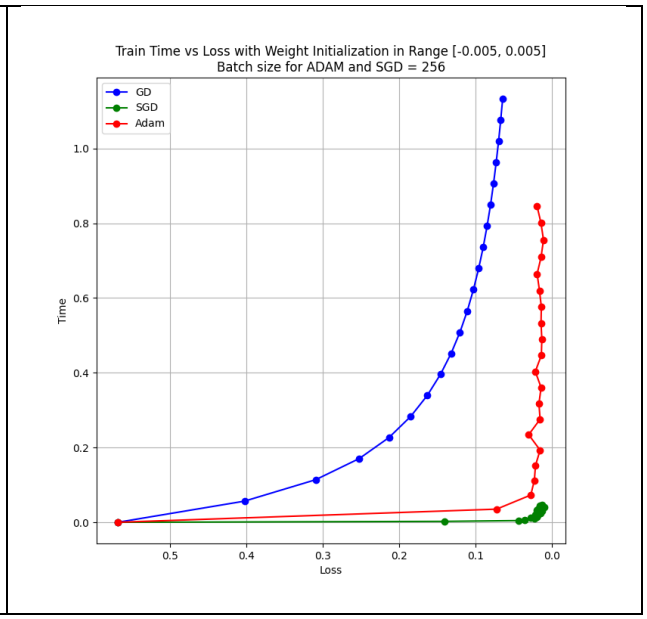

Train Accuracy vs Epoch with Weight Initialization in Range [-0.005, 0.005]
Batch size for ADAM and SGD = 256
GD
SGD
Adam
Accuracy
Epoch


Train Accuracy vs Time with Weight Initialization in Range [-0.005, 0.005]
Batch size for ADAM and SGD = 256
GD
SGD
Adam
Accuracy
Time (Seconds)


Train Epoch vs Loss with Weight Initialization in Range [-0.005, 0.005]
Batch size for ADAM and SGD = 256
GD
SGD
Adam
Epoch
Loss


Train Loss vs Epoch with Weight Initialization in Range [-0.005, 0.005]
Batch size for ADAM and SGD = 256
GD
SGD
Adam
Loss
Epoch


Train Loss vs Time with Weight Initialization in Range [-0.005, 0.005]
Batch size for ADAM and SGD = 256
GD
SGD
Adam
Loss
Time (Seconds)


Train Time vs Loss with Weight Initialization in Range [-0.005, 0.005]
Batch size for ADAM and SGD = 256
GD
SGD
Adam
Time
Loss