Yıldız Teknik Üniversitesi

Bilgisayar Mühendisliği Bölümü

Optimizasyon Projesi Raporu

Ata BAŞ

23011091

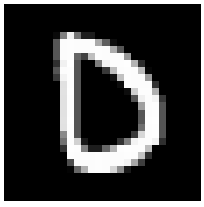
ata.bas@std.yildiz.edu.tr

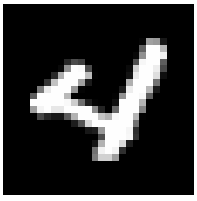
https://drive.google.com/drive/folders/19NtlQlKbJ56ZxAS8yHMQgJvSKhk5yX-S?usp=drive_link

Ön Bilgi

Ödev Konusu: Optimizasyon Algoritmalarını Karşılaştırması

Bu projenin amacı Gradient Descent, Stokastik Gradient Descent, Adam algoritmalarının bir görüntü sınıflandırma probleminde performanslarının kıyaslanması ve yorumlanmasıdır. Analiz, her epoch için model doğruluğu, kayıp değerleri (eğitim ve test), eğitim süreleri ve doğruluk oranları gibi kriterler üzerinden yapılmıştır. Proje iki çıkışlı model üzerinde yapılmıştır.Görüntüler 0 ve 4 grayscale görüntülerinden oluşmaktadır. 2000 adet ‘sıfır rakamı’ ve 2000 adet ‘dört rakamı’ görüntüleri üzerinde eğitim yapılmıştır. Test kümesinde ise 500’er adet test görüntüsü bulunmaktadır.Program ‘sıfır’ ve ‘dört’ görüntülerini tanır ve ayırt eder.





Kodun Açıklanması ve Analizi

.Rastgele “w “ Değeri Oluşturma (random_for_w) :

Ağırlıklar , (w) rastgele değerler atanarak başlatılmıştır. Bu fonksiyon, optimizasyon algoritmalarının farklı başlangıç noktalarından çalışmasını sağlar. Bu süreç , ağırlık çeşitliliğinin de artmasını sağlar.

.w[i] = ağırlık vektörü

```
16  void random_for_w(float *w, int size) {
17      int i;
18      for (i = 0; i < size; i++) {
19          w[i] = ((float)rand() / RAND_MAX) * 2 - 1;
20      }
21  }
```

.Eğitim Görüntülerini Dosyaya Kaydetme (log_to_file) :

Eğitim görüntülerinin optimizasyon algoritmalarıyla işlenmesi sonucunda oluşan iterasyon, loss ve işlem süresi verilerinin dosyalara kaydedilmesini sağlar.

.cost = kayıp .time_elapsed = geçen süre

```
25  void log_to_file(const char *filename, int iteration, float cost, float time_elapsed) {
26      FILE *file = fopen(filename, "a");
27      if (file == NULL) {
28          printf("!Cannot Open The File! File Name: %s\n", filename);
29          exit(-1);
30      }
31      fprintf(file, "%d %.6f %.6f\n", iteration, cost, time_elapsed);
32      fclose(file);
33  }
```

.“w” Değerlerini Kaydetme (save_weights_with_iterations_to_file):

Ağırlıklar her epoch sonunda dosyaya kaydedilir. Bu işlem, optimizasyon sürecindeki değişikliklerin analiz edilmesine yardımcı olur

```c
void save_weights_with_iterations_to_file(const char *filename, float *weights, int size,
    FILE *file = fopen(filename, "a");
    if (file == NULL) {
        printf("!Cannot Open The File! File Name: %s\n", filename);
        exit(-1);
    }
    fprintf(file, "Iteration %d: ", iteration);
    int i;
    for (i = 0; i < size; i++) {
        fprintf(file, "%.6f ", weights[i]);
    }
    fprintf(file, "\n");
    fclose(file);
```

.Görüntüyü İşleme ve Dosyadan Alma (process_image):

Eğitim kümesindeki görüntüler bitmap formunda alınarak işleme sokulur.
Piksel değerleri (0-255) 0-1 aralığına dönüştürülür.
NxN şeklindeki bir görüntü, düz bir vektör haline getirilir.
Her düz vektörün sonuna bir adet bias değeri (1.0) eklenir.

Dosya Açma: `process_image` fonksiyonu, belirtilen dosyayı ikili okuma modunda (`"rb"`) açar.

Bitmap Başlığı (Header): Dosyanın ilk 54 byte'lık kısmını "`header`" dizisine okur. Bu byte'lar, BMP dosyasına ait genişlik, yükseklik ve renk derinliği gibi verilerini içerir.

Görüntü Boyutları ve Bit Derinliği:"`header`" dizisindeki belirli konumlardan (18, 22, 28) sırasıyla görüntünün genişliği, yüksekliği ve bit derinliği gibi değerlerini çıkarır.
Renk Tablosu (Color Table):Eğer bit derinliği 8 veya daha küçükse, renk tablosu okunur. Düşük bit derinliğine sahip BMP dosyaları genellikle 1024 byte'a kadar (256 giriş × 4 byte) bir renk tablosu içerir.
Görüntü Verileri:Görüntü boyutu, `width` (genişlik) ve `height` (yükseklik) değerlerine göre hesaplanır.Görüntü verileri için hafıza (`buf`) ayrılır ve bu veriler dosyadan okunur.

Kod:

```c
float* process_image(const char *filename, int *vector_size) {
    FILE *file = fopen(filename, "rb");
    if (file == NULL) {
        printf("!Cannot Open The File! File Name: %s\n", filename);
        return NULL;
    }

    unsigned char header[54];
    fread(header, sizeof(unsigned char), 54, file);

    int width = *(int *)&header[18];
    int height = *(int *)&header[22];
    int bitDepth = *(int *)&header[28];

    if (bitDepth <= 8) {
        unsigned char colorTable[1024];
        fread(colorTable, sizeof(unsigned char), 1024, file);
    }

    int imageSize = width * height;
    unsigned char *buf = (unsigned char *)malloc(imageSize);
    fread(buf, sizeof(unsigned char), imageSize, file);
    fclose(file);

    *vector_size = imageSize + 1;
    float *vector = (float *)malloc((*vector_size) * sizeof(float));

    int i;
    for (i = 0; i < imageSize; i++) {
        vector[i] = buf[i] / 255.0;
    }
    vector[imageSize] = 1.0;

    free(buf);
    return vector;
}

```

.Test Görüntülerinin Verilerini Yükleme (log_image_test):

Test setinden bir görüntü dosyası okunur. “process_image” fonksiyonu ile okunan görüntü işlenir ve düz bir vektör haline getirilir. Verilen model ağırlığı ile test görüntüsü çarpılır. Elde edilen sonuç, model fonksiyonunda (tanh(w.x)) hesaplanır. Test setinden çekilen görüntünün iterasyonu, loss , işlem süresi verilerinin dosyalara kaydedilmesini sağlar.

Gradient Descent (gradient_descent) :

float **vector_a = “sıfır” eğitim kümesininden alınan görüntü vektörleri

float **vector_b = “dört” eğitim kümesininden alınan görüntü vektörleri

int size = vektörün boyutu

int num_a = “sıfır” eğitim kümesinindeki görüntü sayısı

int num_b = “dört” eğitim kümesinindeki görüntü sayısı

Yöntemin ve Fonksiyonun Açıklanması:

.Gradyan vektörü `gradient` sıfır olarak başlatılır.Eğitim süresi ölçümü için bir zamanlayıcı başlatılır.Döngü, belirlenen `iterations` sayısı kadar tekrarlanır.

.Eğitim görüntülerini, etiketlendirmek için “zero_train” ve “four_train” olmak üzere iki ayrı dosyadan görüntüler çekilir. İlk eğitim kümesinin dosyasından çekilen görüntüler “vector_a[sample][i]” adlı iki boyutlu diziye atanır. İkinci eğitim kümesinin dosyasından çekilen görüntüler “vector_b[sample][i]” adlı iki boyutlu diziye atanır.

.Çekilen bir görüntü rastgele atanmış “w” değeri çarpılar “wx” değişkenine atanır. (wx += w[i] * vector_a[sample][i])

.Elde edilen “wx” , model fonksiyonu olan tanh ile hesaplanır ve sonuç output değişkenine atanır. (output = tanh(wx))

.Output değeri , etikenlenmiş görüntünün eğitilmesi için gerçek değerden beklenen değeri çıkartarak bulunur ve MSE için error’ un karesi alınır ve “cost” ’a atanır.(error = output - 1.0) [cost += pow(error, 2)]

$$\text{Cost} = \frac{1}{N}\sum_{i=1}^{N}(y_{\text{pred}} - y_{\text{true}})^2$$

.Görüntülerin sırayla gradyanları bulunur ve toplanır.
(gradient[i] += (1 - output * output) * error * vector_a[sample][i] * 2.0f / (num_a + num_b))

$$\frac{2}{N} \sum_{i=1}^{\text{num\_a}} (y_{\text{pred}} - y_{\text{true}}) \cdot (1 - y_{\text{pred}}^2) \cdot x_{i,j}$$

.Gradient Descent metotu uygulayarak optimum noktaya ulaşmaya çalışır.
“w” ‘ yi learning rate ve gradyan değeri kullanarak günceller .
(w[i] -= learning_rate * gradient[i])

$$w_j = w_j - \eta \cdot \frac{\partial \text{Cost}}{\partial w_j}$$

Kod:

```
121  void gradient_descent(float *w, float **vector_a, int num_a, float **vector_b, int num_b,
122  int size, int iterations, float learning_rate) {
123      float *gradient = (float *)malloc(size * sizeof(float));
124      clock_t start = clock();
125      int iter;
126      for (iter = 0; iter <= iterations; iter++) {
127          float cost = 0.0;
128          int i;
129          for (i = 0; i < size; i++) {
130              gradient[i] = 0.0;
131          }
132          int sample;
133          for (sample = 0; sample < num_a; sample++) {
134              float wx = 0.0;
135              for (i = 0; i < size; i++) {
136                  wx += w[i] * vector_a[sample][i];
137              }
138              float output = tanh(wx);
139              float error = output - 1.0;
140              cost += pow(error, 2);
141              for (i = 0; i < size; i++) {
142                  gradient[i] += (1 - output * output) * error * vector_a[sample][i] * 2.0f / (num_a + num_b);
143              }
144          }
145
```

```
        for (sample = 0; sample < num_b; sample++) {
            float wx = 0.0;
            for (i = 0; i < size; i++) {
                wx += w[i] * vector_b[sample][i];
            }
            float output = tanh(wx);
            float error = output - (-1.0);
            cost += pow(error, 2);
            for (i = 0; i < size; i++) {
                gradient[i] += (1 - output * output) * error * vector_b[sample][i] * 2.0f / (num_a + num_b);
            }
        }

        cost = cost / (num_a + num_b);
        clock_t end = clock();
        float time_elapsed = (float)(end - start) / CLOCKS_PER_SEC;
        log_to_file("gd_results.txt", iter, cost, time_elapsed);

        for (i = 0; i < size; i++) {
            w[i] -= learning_rate * gradient[i];
        }

        if (iter % 10 == 0) {
            printf("GD Iteration: %d, Cost: %.6f\n", iter, cost);

        }
        log_first_image_test("gd_first_image.txt", iter, cost, time_elapsed, w, "zerotest/zero (1).bmp", size);
        save_weights_with_iterations_to_file("gd_weights_with_iters.txt", w, size, iter);
    }
    free(gradient);
}
```

Stochastic Gradient Descent (mini_batch_sgd) :

float **vector_a = “sıfır” eğitim kümesininden alınan görüntü vektörleri

float **vector_b = “dört” eğitim kümesininden alınan görüntü vektörleri

int size = vektörün boyutu

int num_a = “sıfır” eğitim kümesinindeki görüntü sayısı

int num_b = “dört” eğitim kümesinindeki görüntü sayısı

int batch_size = mini batch boyutu

Yöntemin ve Fonksiyonun Açıklanması:

.Gradient Descent algoritmasının bir varyasyonu olan SGD, her iterasyonda tam veri seti yerine rastgele seçilmiş bir örnek veya küçük bir alt küme (**mini-batch**) üzerinde çalışır. Bu özellik, daha hızlı ve daha verimli çalışmasını sağlar.

.Mini-batch için eğitim kümesinden seçilen rastgele 32 görüntü kullanarak küçük bir grup oluşturulur.

```
for (sample = 0; sample < batch_size / 2; sample++) {
    float wx = 0.0;
    minibatch = (rand() % num_a);
    for (i = 0; i < size; i++) {
        wx += w[i] * vector_a[minibatch][i];
    }
```

Kodun Tamamı:

```
178  void mini_batch_sgd(float *w, float **vector_a, int num_a, float **vector_b, int num_b, int size,
179   int iterations, float learning_rate, int batch_size) {
180      float *gradient = (float *)malloc(size * sizeof(float));
181      clock_t start = clock();
182      int iter;
183      for (iter = 0; iter <= iterations; iter++) {
184          float cost = 0.0;
185          int i;
186          for (i = 0; i < size; i++) {
187              gradient[i] = 0.0;
188          }
189          int minibatch;
190          int sample;
191          for (sample = 0; sample < batch_size / 2; sample++) {
192              float wx = 0.0;
193              minibatch = (rand() % num_a);
194              for (i = 0; i < size; i++) {
195                  wx += w[i] * vector_a[minibatch][i];
196              }
197              float output = tanh(wx);
198              float error = output - 1.0;
199              cost += pow(error, 2);
200              for (i = 0; i < size; i++) {
201                  gradient[i] += (1 - output * output) * error * vector_a[minibatch][i] * 2.0f / (batch_size);
202              }
203          }
```

```
205          for (sample = 0; sample < batch_size / 2; sample++) {
206              float wx = 0.0;
207              minibatch = (rand() % num_b);
208              for (i = 0; i < size; i++) {
209                  wx += w[i] * vector_b[minibatch][i];
210              }
211              float output = tanh(wx);
212              float error = output - (-1.0);
213              cost += pow(error, 2);
214              for (i = 0; i < size; i++) {
215                  gradient[i] += (1 - output * output) * error * vector_b[minibatch][i] * 2.0f / (batch_size);
216              }
217          }
218
219          cost = cost / (batch_size);
220          clock_t end = clock();
221          float time_elapsed = (float)(end - start) / CLOCKS_PER_SEC;
222          log_to_file("sgd_results.txt", iter, cost, time_elapsed);
223
224          for (i = 0; i < size; i++) {
225              w[i] -= learning_rate * gradient[i];
226          }
227
228          if (iter % 10 == 0) {
229              printf("SGD Iteration: %d, Cost: %.6f\n", iter, cost);
230
231
232          }
233          log_first_image_test("sgd_first_image.txt", iter, cost, time_elapsed, w, "zerotest/zero (1).bmp", size);
234          save_weights_with_iterations_to_file("sgd_weights_with_iters.txt", w, size, iter);
235      }
236      free(gradient);
237  }
```

ADAM(adam_optimizer) :

float **vector_a = “sıfır” eğitim kümesininden alınan görüntü vektörleri

float **vector_b = “dört” eğitim kümesininden alınan görüntü vektörleri

int size = vektörün boyutu

int num_a = “sıfır” eğitim kümesinindeki görüntü sayısı

int num_b = “dört” eğitim kümesinindeki görüntü sayısı

Yöntemin ve Fonksiyonun Açıklanması:

Adam **,** geçmiş gradyanları ve gradyanların karelerini kullanarak parametre güncellemeleri yapar. Gradient Descent ‘ten farkı, her parametre için geçmişteki gradyanları ve karelerini dikkate almasıdır. Bundan dolayı GD kodundan tek farkı ağırlık değerlerini güncellediği kısımdadır.

.Hareketli ortalama (m) ve kare ortalama (v) sıfır vektör olarak başlatılır.

Momentum ve Gradyan Karelerini Güncelleme:
Hareketli ortalama (mt) ve kare ortalama (vt) hesaplanır:

```
m[i] = beta1 * m[i] + (1 - beta1) * gradient[i];
v[i] = beta2 * v[i] + (1 - beta2) * gradient[i] * gradient[i];
```

$m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ (Update biased first moment estimate)
$v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$ (Update biased second raw moment estimate)
$\widehat{m}_t \leftarrow m_t / (1 - \beta_1^t)$ (Compute bias-corrected first moment estimate)
$\widehat{v}_t \leftarrow v_t / (1 - \beta_2^t)$ (Compute bias-corrected second raw moment estimate)
$\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \widehat{m}_t / (\sqrt{\widehat{v}_t} + \epsilon)$ (Update parameters)

Bias Correction ve Güncelleme:
.m^t ve v^t hesaplanır ve ağırlıklar güncellenir.

```
w[i] -= learning_rate * m_hat / (sqrt(v_hat) + epsilon);
```

Kod:

```
239  void adam_optimizer(float *w, float **vector_a, int num_a, float **vector_b, int num_b,
240   int size, int iterations, float learning_rate) {
241      float beta1 = 0.9;
242      float beta2 = 0.999;
243      float epsilon = 1e-8;
244
245      float *m = (float *)calloc(size, sizeof(float));
246      float *v = (float *)calloc(size, sizeof(float));
247      float *gradient = (float *)malloc(size * sizeof(float));
248
249      int iter;
250      clock_t start = clock();
251      for (iter = 0; iter <= iterations; iter++) {
252          float cost = 0.0;
253          int i;
254          for (i = 0; i < size; i++) {
255              gradient[i] = 0.0;
256          }
257          int sample;
258          for (sample = 0; sample < num_a; sample++) {
259              float wx = 0.0;
260              for (i = 0; i < size; i++) {
261                  wx += w[i] * vector_a[sample][i];
262              }
263              float output = tanh(wx);
264              float error = output - 1.0;
265              cost += pow(error, 2);
266              for (i = 0; i < size; i++) {
267                  gradient[i] += (1 - output * output) * error * vector_a[sample][i] * 2.0f / (num_a + num_b);
268              }
269          }
```

```
271         for (sample = 0; sample < num_b; sample++) {
272             float wx = 0.0;
273             for (i = 0; i < size; i++) {
274                 wx += w[i] * vector_b[sample][i];
275             }
276             float output = tanh(wx);
277             float error = output - (-1.0);
278             cost += pow(error, 2);
279             for (i = 0; i < size; i++) {
280                 gradient[i] += (1 - output * output) * error * vector_b[sample][i] * 2.0f / (num_a + num_b);
281             }
282         }
283 
284         for (i = 0; i < size; i++) {
285             m[i] = beta1 * m[i] + (1 - beta1) * gradient[i];
286             v[i] = beta2 * v[i] + (1 - beta2) * gradient[i] * gradient[i];
287 
288             float m_hat = m[i] / (1 - pow(beta1, iter + 1));
289             float v_hat = v[i] / (1 - pow(beta2, iter + 1));
290 
291             w[i] -= learning_rate * m_hat / (sqrt(v_hat) + epsilon);
292         }
293 
294         cost = cost / (num_a + num_b);
295         clock_t end = clock();
296         float time_elapsed = (float)(end - start) / CLOCKS_PER_SEC;
297         log_to_file("adam_results.txt", iter, cost, time_elapsed);
298         save_weights_with_iterations_to_file("adam_weights_with_iters.txt", w, size, iter);
299 
300         if (iter % 10 == 0) {
301             printf("Adam Iteration: %d, Cost: %.6f\n", iter, cost);
302 
303 
304         }
305         log_first_image_test("adam_first_image.txt", iter, cost, time_elapsed, w, "zerotest/zero (1).bmp", size);
306     }
```

Görüntü Tahmini (classify_images) :

Yöntemin ve Fonksiyonun Açıklanması:

.Verilen görüntü vektörünü, “weight” vektörü ile çarpıp çıkan sonucu model fonksiyonuna (tanh) koyar.

.Çıkan sonuca göre 0’dan büyük ise “1”(sıfır) , 0’dan küçük ise “-1”(dört) tahmininde bulunur.

. ”predition=1” ve görüntü gerçekten ”sıfır resmi” ise beklenen değer ile tahmin birbirine eşit çıkar.”predition=0” ve görüntü gerçekten ”dört resmi” ise beklenen değer ile tahmin birbirine eşit çıkar.

```
int classify_images(const char *folder_path, float *weights, int vector_size, const char *label,
int expected_label) {
    int correct_predictions = 0;
    int img;
    printf("Processing folder: %s\n", folder_path);
    for (img = 1; img <= TEST_IMAGE_COUNT; img++) {
        char filepath[512];
        snprintf(filepath, sizeof(filepath), "%s/%s (%d).bmp", folder_path, label, img);

        float *test_vector = process_image(filepath, &vector_size);
        if (test_vector) {
            float wx = 0.0;
            int i;
            for (i = 0; i < vector_size; i++) {
                wx += weights[i] * test_vector[i];
            }
            int prediction;
            if(tanh(wx) > 0){
                prediction = 1;
            }
            else{
                prediction = -1;
            }

            if (prediction == expected_label) {
                correct_predictions++;
            }
            free(test_vector);
        }
    }
    return correct_predictions;
}
```

Ortalama “w” Değeri Hesaplama

(calculate_average_weights) :

.Tüm w değerlerinin ortalaması alınarak tek bir w değeri bulunur.

```
void calculate_average_weights(float **weights, float *average_weights, int num_weights, int vector_size) {
    int i,j;
    for (i = 0; i < vector_size; i++) {
        average_weights[i] = 0.0;
        for (j = 0; j < num_weights; j++) {
            average_weights[i] += weights[j][i];
        }
        average_weights[i] /= num_weights;
    }
}
```

5 farklı ilk “w” değeri :

```c
int i;
for (i = 0; i < NUM_WEIGHTS; i++) {
    weights_set_gd[i] = (float *)malloc(vector_size * sizeof(float));
    random_for_w(weights_set_gd[i], vector_size);

    printf("\nGD Training for Weight Set %d\n", i + 1);
    gradient_descent(weights_set_gd[i], vector_a, NUM_IMAGES, vector_b, NUM_IMAGES, vector_size, ITERATIONS, LEARNING_RATE);
}

for (i = 0; i < NUM_WEIGHTS; i++) {
    weights_set_sgd[i] = (float *)malloc(vector_size * sizeof(float));
    random_for_w(weights_set_sgd[i], vector_size);

    printf("\nMini-Batch SGD Training for Weight Set %d\n", i + 1);
    mini_batch_sgd(weights_set_sgd[i], vector_a, NUM_IMAGES, vector_b, NUM_IMAGES, vector_size, 200, LEARNING_RATE, MINI_BATCH_SIZE);
}

for (i = 0; i < NUM_WEIGHTS; i++) {
    weights_set_adam[i] = (float *)malloc(vector_size * sizeof(float));
    random_for_w(weights_set_adam[i], vector_size);

    printf("\nAdam Training for Weight Set %d\n", i + 1);
    adam_optimizer(weights_set_adam[i], vector_a, NUM_IMAGES, vector_b, NUM_IMAGES, vector_size, ITERATIONS, LEARNING_RATE);
}
```

Gradient Descent Output:

```
Model Training Started.

GD Training for Weight Set 1
GD Iteration: 0, Cost: 1.784429
GD Iteration: 10, Cost: 0.950740
GD Iteration: 20, Cost: 0.627885
GD Iteration: 30, Cost: 0.454109
GD Iteration: 40, Cost: 0.365122
GD Iteration: 50, Cost: 0.307502
GD Iteration: 60, Cost: 0.266425
GD Iteration: 70, Cost: 0.235487
GD Iteration: 80, Cost: 0.210435
GD Iteration: 90, Cost: 0.189582
GD Iteration: 100, Cost: 0.171741

GD Training for Weight Set 2
GD Iteration: 0, Cost: 2.251978
GD Iteration: 10, Cost: 1.824772
GD Iteration: 20, Cost: 1.558005
GD Iteration: 30, Cost: 1.214692
GD Iteration: 40, Cost: 0.886090
GD Iteration: 50, Cost: 0.678075
GD Iteration: 60, Cost: 0.553250
GD Iteration: 70, Cost: 0.472769
GD Iteration: 80, Cost: 0.416980
GD Iteration: 90, Cost: 0.375929
GD Iteration: 100, Cost: 0.342748

GD Training for Weight Set 3
GD Iteration: 0, Cost: 1.495891
GD Iteration: 10, Cost: 0.623045
GD Iteration: 20, Cost: 0.305887
GD Iteration: 30, Cost: 0.232768
GD Iteration: 40, Cost: 0.192359
GD Iteration: 50, Cost: 0.163415
GD Iteration: 60, Cost: 0.141607
GD Iteration: 70, Cost: 0.125437
GD Iteration: 80, Cost: 0.113346
GD Iteration: 90, Cost: 0.103966
GD Iteration: 100, Cost: 0.096385
```

```
GD Training for Weight Set 4
GD Iteration: 0, Cost: 1.545601
GD Iteration: 10, Cost: 0.952977
GD Iteration: 20, Cost: 0.634263
GD Iteration: 30, Cost: 0.480608
GD Iteration: 40, Cost: 0.394266
GD Iteration: 50, Cost: 0.343004
GD Iteration: 60, Cost: 0.305694
GD Iteration: 70, Cost: 0.274278
GD Iteration: 80, Cost: 0.248579
GD Iteration: 90, Cost: 0.227790
GD Iteration: 100, Cost: 0.210838

GD Training for Weight Set 5
GD Iteration: 0, Cost: 1.604020
GD Iteration: 10, Cost: 0.857891
GD Iteration: 20, Cost: 0.483902
GD Iteration: 30, Cost: 0.334171
GD Iteration: 40, Cost: 0.261193
GD Iteration: 50, Cost: 0.213831
GD Iteration: 60, Cost: 0.183367
GD Iteration: 70, Cost: 0.164169
GD Iteration: 80, Cost: 0.150083
GD Iteration: 90, Cost: 0.138598
GD Iteration: 100, Cost: 0.128698
```

Stochastic Gradient Descent Output:

```
Mini-Batch SGD Training for Weight Set 1
SGD Iteration: 0, Cost: 2.633776
SGD Iteration: 10, Cost: 1.549160
SGD Iteration: 20, Cost: 1.923715
SGD Iteration: 30, Cost: 1.175226
SGD Iteration: 40, Cost: 1.224880
SGD Iteration: 50, Cost: 0.900484
SGD Iteration: 60, Cost: 0.233030
SGD Iteration: 70, Cost: 1.019160
SGD Iteration: 80, Cost: 0.553359
SGD Iteration: 90, Cost: 0.313253
SGD Iteration: 100, Cost: 0.142664
SGD Iteration: 110, Cost: 0.047192
SGD Iteration: 120, Cost: 0.089537
SGD Iteration: 130, Cost: 0.000684
SGD Iteration: 140, Cost: 0.068465
SGD Iteration: 150, Cost: 0.138275
SGD Iteration: 160, Cost: 0.157912
SGD Iteration: 170, Cost: 0.077082
SGD Iteration: 180, Cost: 0.239862
SGD Iteration: 190, Cost: 0.142491
SGD Iteration: 200, Cost: 0.000187

Mini-Batch SGD Training for Weight Set 2
SGD Iteration: 0, Cost: 2.386906
SGD Iteration: 10, Cost: 2.001371
SGD Iteration: 20, Cost: 1.999915
SGD Iteration: 30, Cost: 1.910000
SGD Iteration: 40, Cost: 2.039497
SGD Iteration: 50, Cost: 1.998371
SGD Iteration: 60, Cost: 1.775202
SGD Iteration: 70, Cost: 1.920903
SGD Iteration: 80, Cost: 1.960478
SGD Iteration: 90, Cost: 1.787709
SGD Iteration: 100, Cost: 1.028083
SGD Iteration: 110, Cost: 1.236116
SGD Iteration: 120, Cost: 0.738397
SGD Iteration: 130, Cost: 0.583112
SGD Iteration: 140, Cost: 0.805975
SGD Iteration: 150, Cost: 0.125983
SGD Iteration: 160, Cost: 0.661273
SGD Iteration: 170, Cost: 0.261910
SGD Iteration: 180, Cost: 0.346052
SGD Iteration: 190, Cost: 0.270084
SGD Iteration: 200, Cost: 0.263785
```

```
Mini-Batch SGD Training for Weight Set 3
SGD Iteration: 0, Cost: 2.106491
SGD Iteration: 10, Cost: 1.952283
SGD Iteration: 20, Cost: 1.915625
SGD Iteration: 30, Cost: 1.452522
SGD Iteration: 40, Cost: 0.382156
SGD Iteration: 50, Cost: 0.986774
SGD Iteration: 60, Cost: 0.111156
SGD Iteration: 70, Cost: 0.449931
SGD Iteration: 80, Cost: 0.006917
SGD Iteration: 90, Cost: 0.201726
SGD Iteration: 100, Cost: 0.063720
SGD Iteration: 110, Cost: 0.199495
SGD Iteration: 120, Cost: 0.000287
SGD Iteration: 130, Cost: 0.240550
SGD Iteration: 140, Cost: 0.001799
SGD Iteration: 150, Cost: 0.124822
SGD Iteration: 160, Cost: 0.000061
SGD Iteration: 170, Cost: 0.213362
SGD Iteration: 180, Cost: 0.003475
SGD Iteration: 190, Cost: 0.216361
SGD Iteration: 200, Cost: 0.124141

Mini-Batch SGD Training for Weight Set 4
SGD Iteration: 0, Cost: 2.558617
SGD Iteration: 10, Cost: 1.672888
SGD Iteration: 20, Cost: 1.233303
SGD Iteration: 30, Cost: 1.064006
SGD Iteration: 40, Cost: 0.362433
SGD Iteration: 50, Cost: 0.902924
SGD Iteration: 60, Cost: 0.281166
SGD Iteration: 70, Cost: 0.298943
SGD Iteration: 80, Cost: 0.001355
SGD Iteration: 90, Cost: 0.149414
SGD Iteration: 100, Cost: 0.351131
SGD Iteration: 110, Cost: 0.000817
SGD Iteration: 120, Cost: 0.125532
SGD Iteration: 130, Cost: 0.323674
SGD Iteration: 140, Cost: 0.012509
SGD Iteration: 150, Cost: 0.227311
SGD Iteration: 160, Cost: 0.087266
SGD Iteration: 170, Cost: 0.072053
SGD Iteration: 180, Cost: 0.247515
SGD Iteration: 190, Cost: 0.038496
SGD Iteration: 200, Cost: 0.080157
```

```
Mini-Batch SGD Training for Weight Set 5
SGD Iteration: 0, Cost: 2.166090
SGD Iteration: 10, Cost: 1.494974
SGD Iteration: 20, Cost: 1.123896
SGD Iteration: 30, Cost: 0.619590
SGD Iteration: 40, Cost: 0.921301
SGD Iteration: 50, Cost: 0.566232
SGD Iteration: 60, Cost: 0.014352
SGD Iteration: 70, Cost: 0.462240
SGD Iteration: 80, Cost: 0.125003
SGD Iteration: 90, Cost: 0.159097
SGD Iteration: 100, Cost: 0.125667
SGD Iteration: 110, Cost: 0.180413
SGD Iteration: 120, Cost: 0.289745
SGD Iteration: 130, Cost: 0.180126
SGD Iteration: 140, Cost: 0.000632
SGD Iteration: 150, Cost: 0.125596
SGD Iteration: 160, Cost: 0.000001
SGD Iteration: 170, Cost: 0.122863
SGD Iteration: 180, Cost: 0.121655
SGD Iteration: 190, Cost: 0.233957
SGD Iteration: 200, Cost: 0.227679
```

ADAM Output:

```
Adam Training for Weight Set 1
Adam Iteration: 0, Cost: 1.897516
Adam Iteration: 10, Cost: 0.070846
Adam Iteration: 20, Cost: 0.040502
Adam Iteration: 30, Cost: 0.030024
Adam Iteration: 40, Cost: 0.020784
Adam Iteration: 50, Cost: 0.017348
Adam Iteration: 60, Cost: 0.016247
Adam Iteration: 70, Cost: 0.016077
Adam Iteration: 80, Cost: 0.016041
Adam Iteration: 90, Cost: 0.016028
Adam Iteration: 100, Cost: 0.016021

Adam Training for Weight Set 2
Adam Iteration: 0, Cost: 1.571545
Adam Iteration: 10, Cost: 0.040578
Adam Iteration: 20, Cost: 0.027520
Adam Iteration: 30, Cost: 0.020142
Adam Iteration: 40, Cost: 0.015015
Adam Iteration: 50, Cost: 0.012704
Adam Iteration: 60, Cost: 0.010873
Adam Iteration: 70, Cost: 0.008939
Adam Iteration: 80, Cost: 0.007109
Adam Iteration: 90, Cost: 0.006423
Adam Iteration: 100, Cost: 0.006048
```

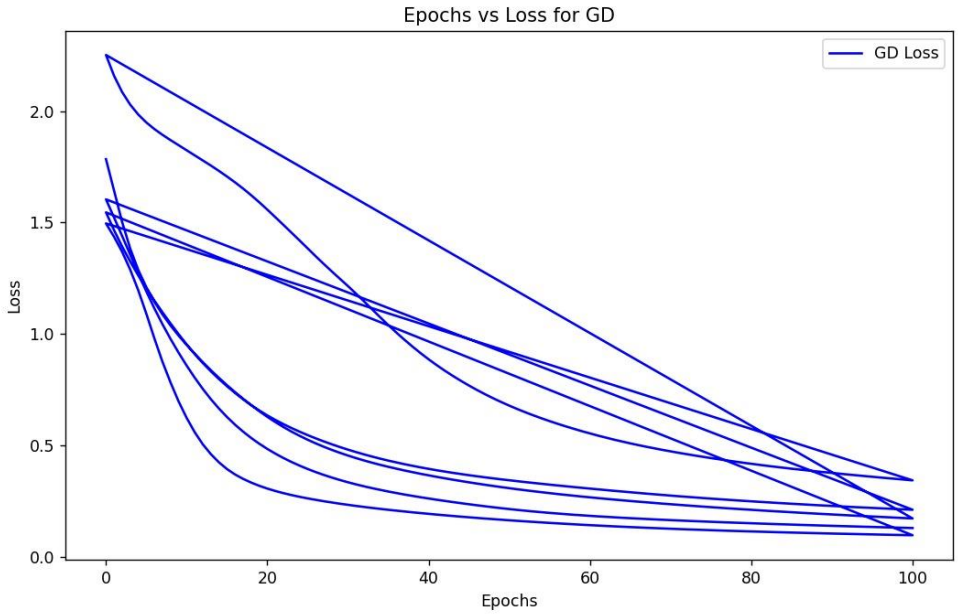
```
Adam Training for Weight Set 3
Adam Iteration: 0, Cost: 2.823308
Adam Iteration: 10, Cost: 0.106942
Adam Iteration: 20, Cost: 0.046588
Adam Iteration: 30, Cost: 0.033150
Adam Iteration: 40, Cost: 0.029240
Adam Iteration: 50, Cost: 0.025280
Adam Iteration: 60, Cost: 0.023759
Adam Iteration: 70, Cost: 0.022685
Adam Iteration: 80, Cost: 0.021497
Adam Iteration: 90, Cost: 0.021001
Adam Iteration: 100, Cost: 0.020881

Adam Training for Weight Set 4
Adam Iteration: 0, Cost: 1.238271
Adam Iteration: 10, Cost: 0.040084
Adam Iteration: 20, Cost: 0.022865
Adam Iteration: 30, Cost: 0.015458
Adam Iteration: 40, Cost: 0.013028
Adam Iteration: 50, Cost: 0.011885
Adam Iteration: 60, Cost: 0.009986
Adam Iteration: 70, Cost: 0.009280
Adam Iteration: 80, Cost: 0.008329
Adam Iteration: 90, Cost: 0.008023
Adam Iteration: 100, Cost: 0.007077

Adam Training for Weight Set 5
Adam Iteration: 0, Cost: 1.753784
Adam Iteration: 10, Cost: 0.063301
Adam Iteration: 20, Cost: 0.027368
Adam Iteration: 30, Cost: 0.021647
Adam Iteration: 40, Cost: 0.018105
Adam Iteration: 50, Cost: 0.015033
Adam Iteration: 60, Cost: 0.013104
Adam Iteration: 70, Cost: 0.012476
Adam Iteration: 80, Cost: 0.010459
Adam Iteration: 90, Cost: 0.009761
Adam Iteration: 100, Cost: 0.009152
```

DOĞRULUK ORANLARI:

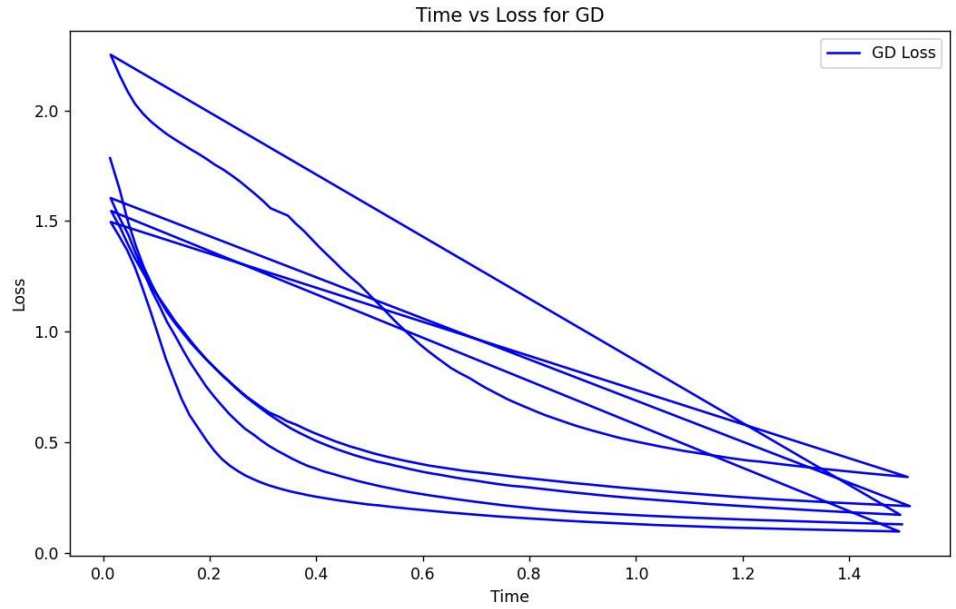
```
Evaluating GD Weights:
Processing folder: zerotest
Processing folder: fourtest

Evaluating Mini-Batch SGD Weights:
Processing folder: zerotest
Processing folder: fourtest

Evaluating Adam Weights:
Processing folder: zerotest
Processing folder: fourtest

GD Total Correct Predictions: 989/1000
GD Accuracy: 98.90%

Mini-Batch SGD Total Correct Predictions: 995/1000
Mini-Batch SGD Accuracy: 99.50%

Adam Total Correct Predictions: 996/1000
Adam Accuracy: 99.60%
```

!Grafiklerde oluşan düz çizgiler, verileri tek dosyasından aldığımız için kaynaklanıyor. İterasyonlar arasındaki çizgileri kaldırmamak için de bu düz çizgileri silemiyoruz!

GRAFİKLER



.X ekseni = iterayon
.Y ekseni = modelin hata oranını
.Daha düşük bir loss değeri, modelin daha iyi performans gösterdiği anlamına gelir.
.Her bir çizgi = farklı bir başlangıç noktası , farklı bir “w” ilk değeri
.Başlangıçta yüksek olan kayıplar, zamanla azalarak daha iyi sonuçlara ulaşıyor.
.Bazı çizgiler çok hızlı düşerken diğerleri daha yavaş bir azalma gösteriyor. Bu durum, local minimum , grafikteki eğim veya başlangıçtaki ağırlıklardan kaynaklanıyor olabilir.
.Eğrilerin sonunda, kaybın minimum seviyeye ulaştığını görebiliyoruz.



.X ekseni = geçen süre

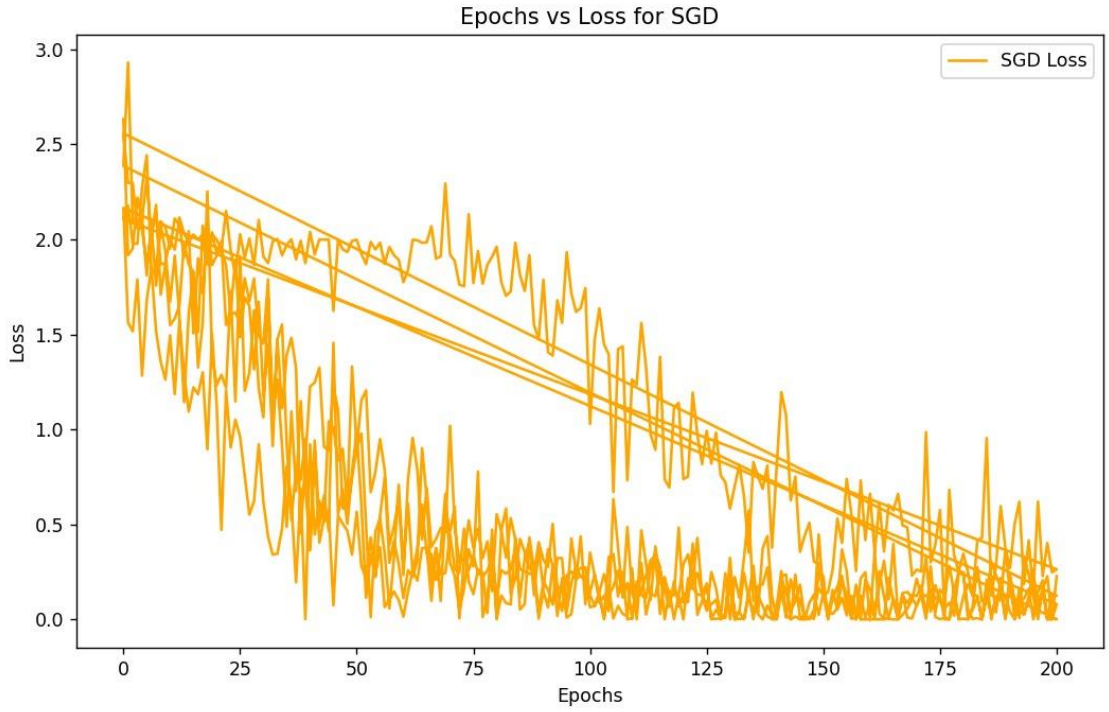
.Y ekseni = modelin hata oranını

.Her bir çizgi = farklı bir başlangıç noktası , farklı bir “w” ilk değeri

.Başlangıçta kayıpların çok yüksek olduğu görülüyor, ancak zamanla model bu kayıpları azaltıyor. Bu durum, algoritmanın belirli bir süre içinde optimizasyonu gerçekleştirdiğini ifade ediyor.

.Ancak bazı çizgilerin zaman ekseninde daha hızlı bir şekilde minimum noktaya ulaştığı görülüyor. Bu da grafikteki local minimum, eğim veya başlangıç noktalarının farklılıklarından kaynaklanabilir.

.Modelin optimizasyon sürecinde zaman açısından verimli çalıştığını gözlemliyoruz. Ancak bazı çizgiler daha fazla zaman harcıyor, bu da yavaş bir öğrenme sürecine işaret edebilir.



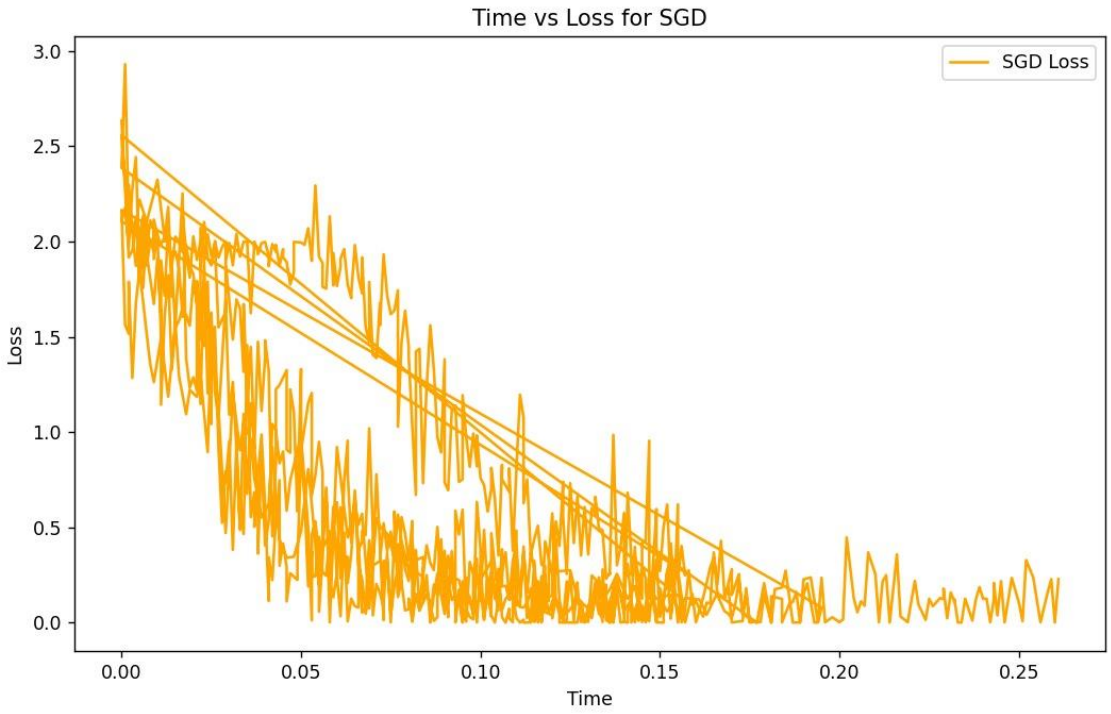
.X ekseni = iterasyon

.Y ekseni = modelin hata oranını

.Grafikte, kayıp oranında başlangıçta yüksek bir düzensizlik ve gürültü mevcut. Bu, SGD'nin (Stochastic Gradient Descent) doğasında bulunan mini-batch öğrenme sürecinin bir etkisidir. SGD, her iterasyonda daha küçük bir veri alt kümesi üzerinde optimizasyon yaptığı için daha fazla varyans gösterir.

.Kaybın genel olarak azaldığı, ancak bu azalma sırasında iniş-çıkışlar olduğu görülüyor. SGD, bazı iterasyonlarda kaybı artırmış gibi görünse de, genel eğilim doğru yönde ilerliyor.

.Grafik, SGD'nin etkili bir şekilde kaybı azalttığını gösteriyor, ancak bu süreç Gradient Descent (GD) algoritmasına kıyasla daha az stabil görünüyor.

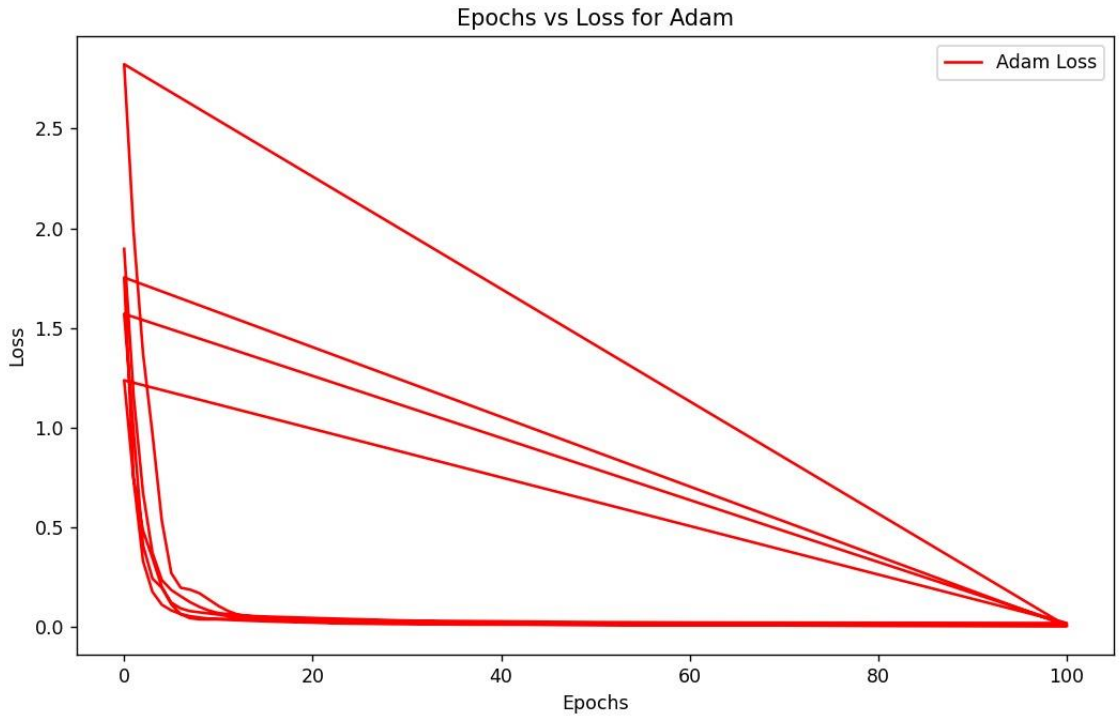


.X ekseni = geçen süre

.Y ekseni = modelin hata oranını

.SGD'nin optimizasyon sürecindeki düzensizlikler burada da belirgin. Ancak genel olarak kaybın zaman içinde sürekli azaldığını görebiliyoruz.

.SGD, toplamda daha kısa sürede minimum kayba ulaşabiliyor. Bu, özellikle büyük veri setleri üzerinde daha verimli olabileceğini gösteriyor. Ancak gürültü nedeniyle her iterasyonda stabil bir iyileşme görülmüyor.



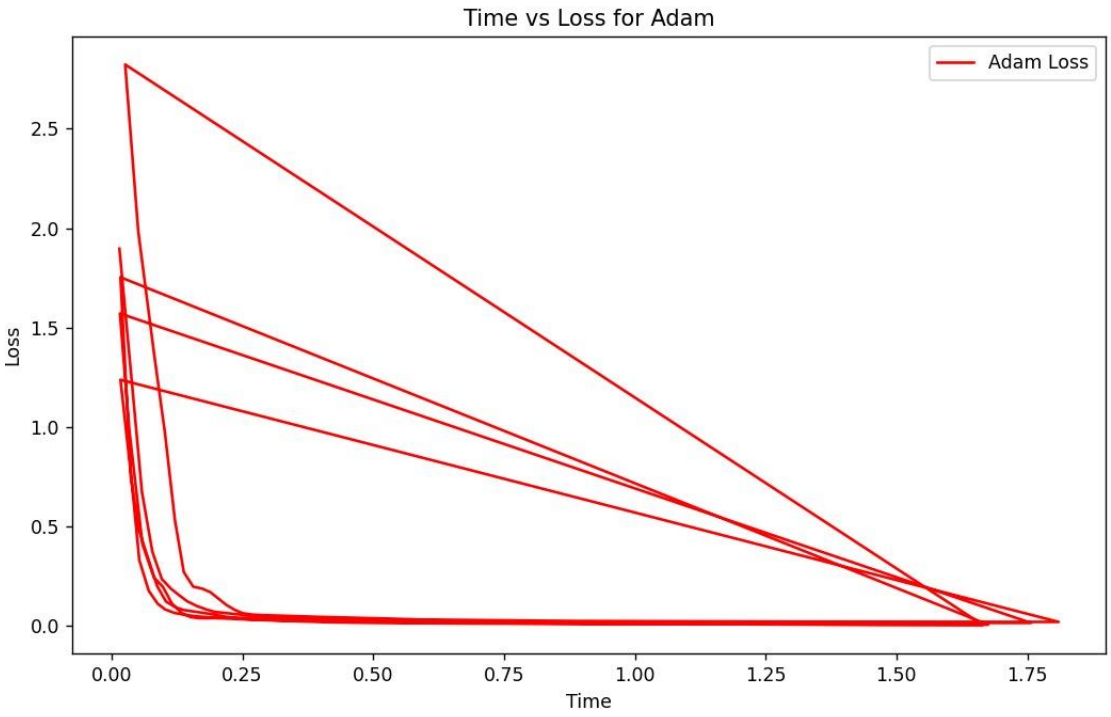
.X ekseni = geçen süre

.Y ekseni = modelin hata oranını

.Başlangıçta kayıp değerleri yüksektir

.İlk birkaç epoch içinde kayıplar çok hızlı bir şekilde düşüyor. Adam algoritmasının hızlı öğrenme kapasitesini bu eğilim doğruluyor.

.Adam algoritması, kaybı hızlı bir şekilde minimize etmiş ve eğitim süreci oldukça etkili gözüküyor.



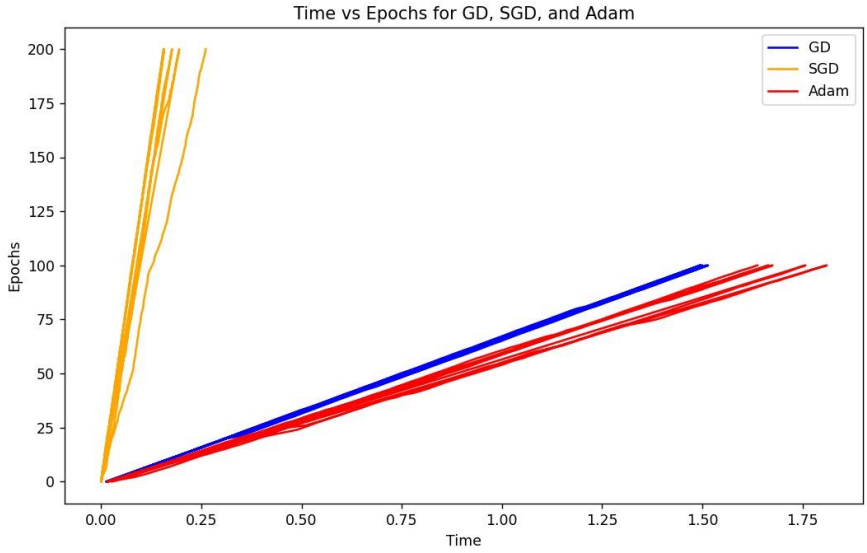
.X ekseni = geçen süre

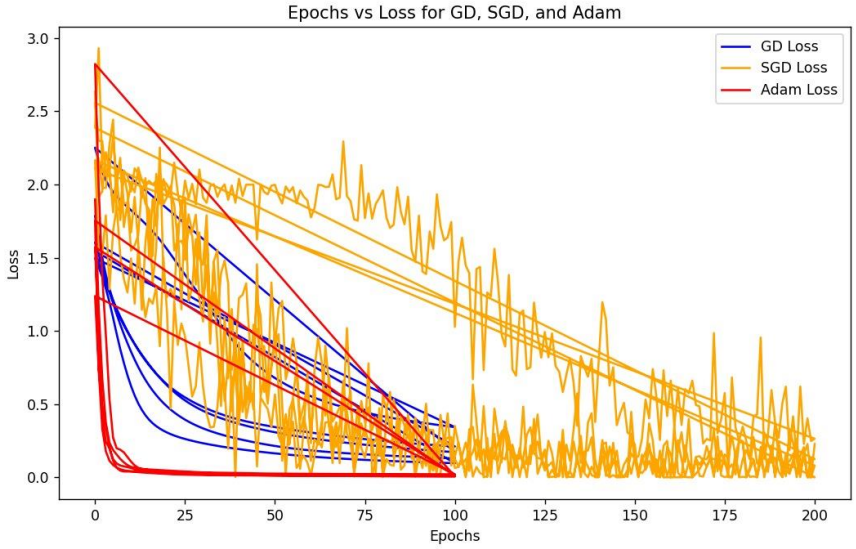
.Y ekseni = modelin hata oranını

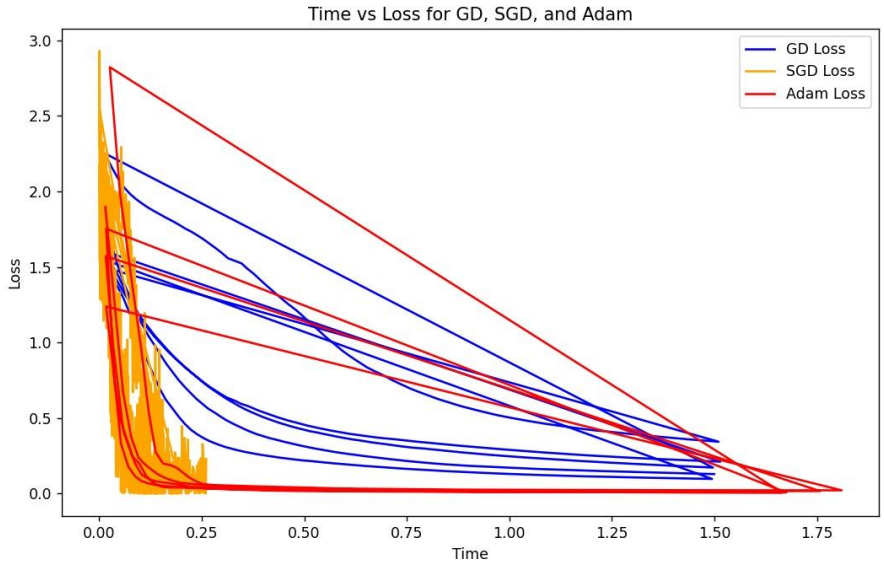
.Kayıp fonksiyonu, kısa bir sürede hızlı bir şekilde sıfıra yaklaşıyor.

.Çizgilerin eğimi, optimizasyonun başlangıçta çok hızlı olduğunu, ancak belirli bir noktadan sonra yavaşladığını gösteriyor.

.Zaman eksenindeki bu davranış, Adam algoritmasının hem verimli hem de kararlı bir öğrenme sunduğunu gösteriyor.


Time vs Epochs for GD, SGD, and Adam
GD
SGD
Adam
Epochs
Time
0
25
50
75
100
125
150
175
200
0.00
0.25
0.50
0.75
1.00
1.25
1.50
1.75


Epochs vs Loss for GD, SGD, and Adam
GD Loss
SGD Loss
Adam Loss
Loss
Epochs
0.0
0.5
1.0
1.5
2.0
2.5
3.0
0
25
50
75
100
125
150
175
200


Time vs Loss for GD, SGD, and Adam
GD Loss
SGD Loss
Adam Loss
Loss
Time
0.0
0.5
1.0
1.5
2.0
2.5
3.0
0.00
0.25
0.50
0.75
1.00
1.25
1.50
1.75

Grafiklerin Yorumlanması ve Karşılaştırması:

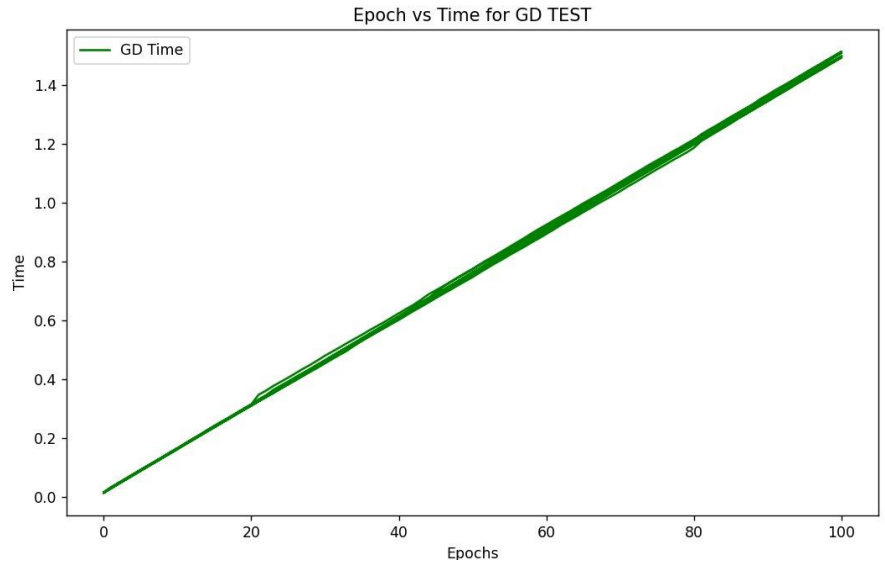
GD , SGD ve Adam ‘ a göre daha yavaştır ama kararlılığı oldukça yüksektir. Hesaplama maliyeti yüksektir . Küçük veri setleri ve modellerde uygun olabilir.

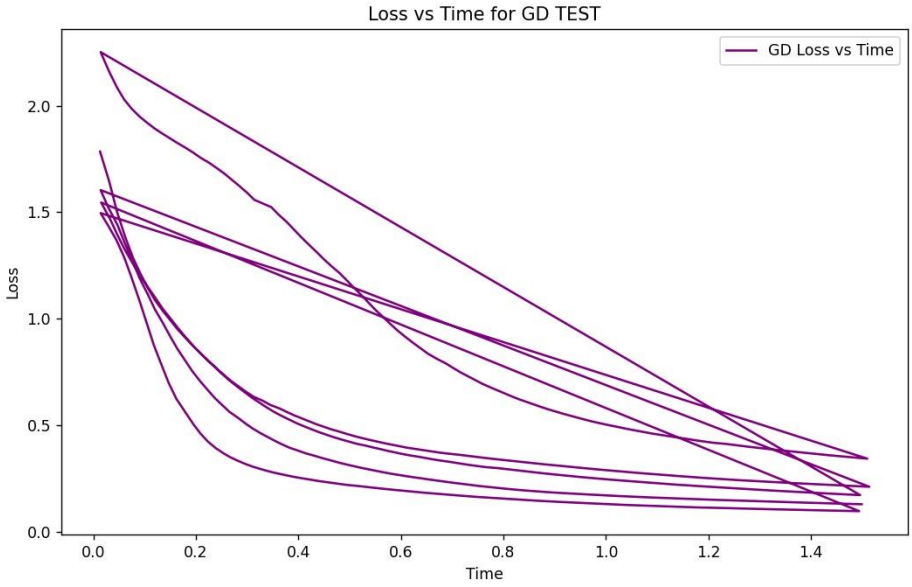
SGD, Adam ve GD ile karşılaştırıldığında çok daha hızlıdır, çünkü her iterasyonda tüm veri setini değil, sadece bir belirli bir kısmını yani mini-batch'i kullanır. Ancak, bunun bir dezavantajı vardır. Rastgele veri seçimi nedeniyle öğrenme sürecinde dalgalanmalar meydana gelir ve optimum noktaya ulaşabilmesi için daha fazla iterasyon gerekir. Yani, SGD’nin her iterasyonu daha düşük maliyetlidir, ancak istenen sonuca ulaşmak için daha fazla iterasyona ihtiyaç duyar. Büyük veri setlerinde yaygın olarak kullanılır

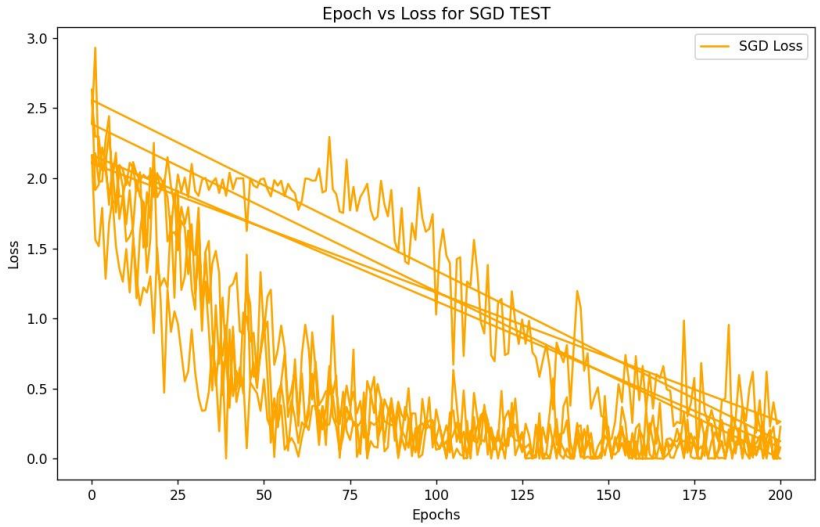
Adam ise, GD’den farklı olarak geçmiş gradyanları dikkate alır, bu sayede daha stabil bir öğrenme sağlar. Yani yerel minimum gibi noktalara takılmadan ilerleyebilir. Adaptif öğrenme sağlar . Gürültülü gradyanlarda bile iyi performans gösterir. Ancak aşırı optimizasyon (overfitting) riski taşıyabilir.

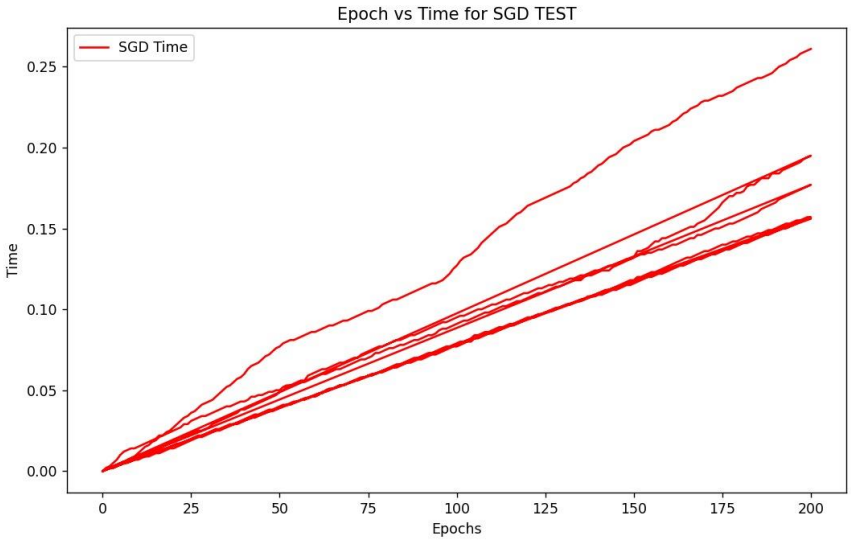
Bu özellikler göz önünde bulundurularak, elimizdeki problem ve kaynaklara bağlı olarak hangi algoritmanın kullanılacağına karar verilebilir.

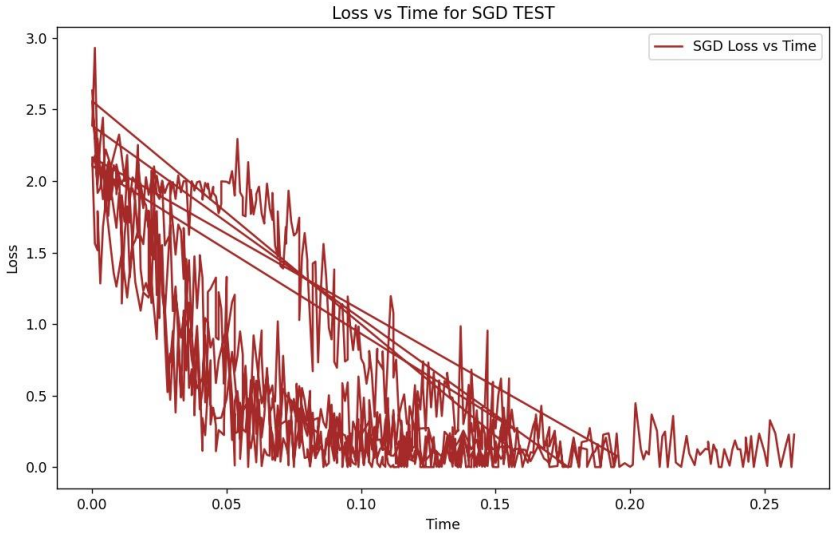
TEST VERİLERİ ÜZERİNDE GRAFİKLER:

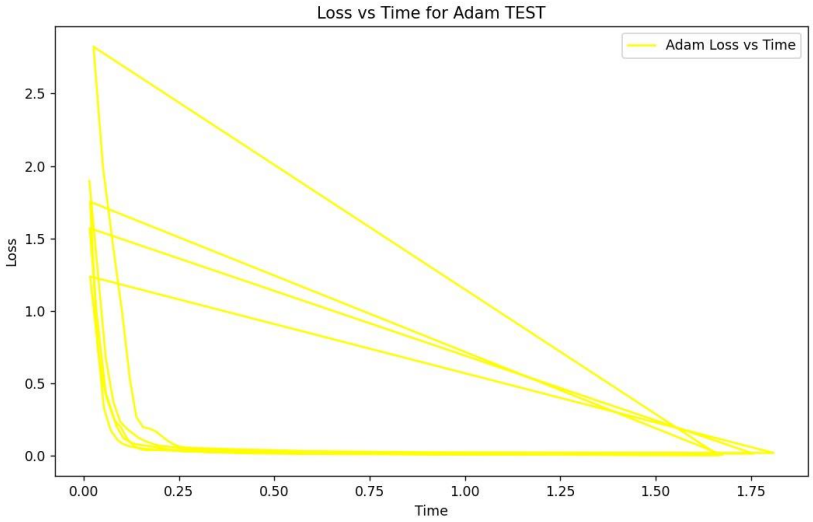












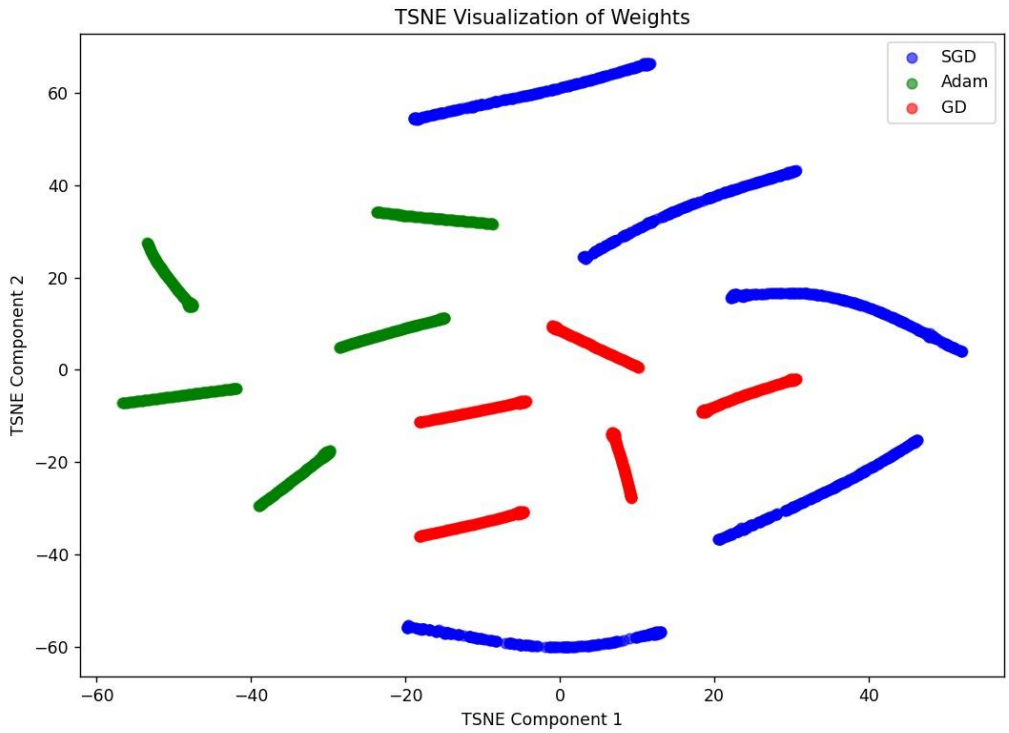
Test Grafikleri Genel Yorum:

Bu grafikler, modelin sadece eğitimde değil, test aşamasında da başarılı olduğunu kanıtlıyor. Eğer test setindeki kayıp değerleri rastgele veya beklenmeyen şekillerde hareket etseydi, modelin overfitting yaptığı ya da test verisiyle iyi çalışmadığı söylenebilirdi. Ancak bu grafikler, test setinde de modelin sağlam bir performans sergilediğini ortaya koyuyor. Test seti grafiğinin eğitim seti grafiği ile benzer bir yapıya sahip olması, modelin genelleme kapasitesini güçlü olduğunu gösteriyor. Farklı "w" değerlerine rağmen, test sürecinde de modelin düzenli bir şekilde performans sergilemesi, bu çalışmanın iyi yürütüldüğünü gösteriyor.

Sonuç olarak bu grafikler, test verisinde modelin öğrenme sürecini tutarlı ve başarılı bir şekilde temsil ettiğini göstermektedir.

B: Optimizasyon Sürecinin 2 Boyutta Gösterimi

- TSNE Yorumu



.t-SNE Bileşen 1 (x-ekseni) ve t-SNE Bileşen 2 (y-ekseni): Bu eksenler, model ağırlıklarının yüksek boyutlu uzaydan düşük boyutlu bir uzaya indirgenmiş temsilini gösteriyor.

SGD (Mavi):

Bu noktalar, genellikle daha geniş bir alana yayılmış ve daha "eğik" kümeler oluşturmuş gibi görünüyor.

Bu, SGD'nin doğası gereği rastgelelik içermesinden kaynaklanabilir (örneğin, minibatch kullanımı nedeniyle).
SGD'nin daha farklı lokal minimumları keşfetme potansiyeline sahip olduğunu gösterebilir.

Adam (Yeşil):

Yeşil noktalar daha kompakt ve odaklanmış kümeler oluşturuyor.
Bu, Adam algoritmasının adaptif öğrenme oranları sayesinde daha kararlı ve kesin bir optimizasyon süreci sunduğunu gösteriyor.
Adam'ın parametre uzayında daha kesin bir şekilde yakınsadığını ima edebilir.

GD (Kırmızı):

Kırmızı noktalar daha belirgin, küçük ve birbirinden ayrı kümeler oluşturmuş.
Bu, Gradient Descent'in genelde daha kararlı ve düzenli bir optimizasyon yolu izlediğini gösterebilir.
Ancak tam gradyan inişi, hesaplama açısından daha pahalı olabilir ve bu nedenle SGD veya Adam kadar esnek olmayabilir.