YILDIZ TEKNİK ÜNİVERSİTESİ

BİLGİSAYAR MÜHENDİSLİĞİ DİFERANSİYEL DENKLEMLER ÖDEVİ

AD-SOYAD : YUSUF BAŞAR GÜNDÜZ

NO : 23011029

KONU : Optimizasyon algoritmalarını karşılaştırma

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <time.h>

#define N 28    //piksel boyutu
#define PIXELS (N * N)
#define DATA_SIZE 2500     //Veri seti boyutu
#define TRAIN_RATIO 0.8    //Eğitim/Test oranı
#define TRAIN_SIZE (int)(DATA_SIZE * TRAIN_RATIO)
#define TEST_SIZE (DATA_SIZE - TRAIN_SIZE)
#define LEARNING_RATE 0.3
#define EPOCHS 100
#define BETA1 0.9
#define BETA2 0.999
#define EPSILON 1e-8
double data1[DATA_SIZE][PIXELS + 1];
double data2[DATA_SIZE][PIXELS + 1];
double labels1[DATA_SIZE];
double labels2[DATA_SIZE];
double weights_gd[PIXELS + 1];      //GD için ağırlıklar
double weights_sgd[PIXELS + 1];     //SGD için ağırlıklar
double weights_adam[PIXELS + 1];    //Adam için ağırlıklar
double m[PIXELS + 1];               //hareketli ortalama
double v[PIXELS + 1];               //kare ortalama
double t = 0;                       //zaman
```

İlk kısımda tanımlamalar ve makrolar kullanlır:

N: Görüntü boyutu (28x28 piksellik görüntüler için N = 28).

PIXELS: Görüntüdeki toplam piksel sayısı (28 * 28 = 784).

DATA_SIZE: Veri kümesindeki toplam veri sayısı (2500).

TRAIN_RATIO: Eğitim ve test verileri arasındaki oran (%80 eğitim, %20 test).

LEARNING_RATE: Öğrenme oranı.

EPOCHS: Eğitim döngüsü sayısı.

BETA1, BETA2, EPSILON: Adam optimizasyonunda kullanılan hiperparametreler.

Data ve Label: Veri setinden okunan piksel ve etiketlerin yeri.

Weights: Üç farklı optimizasyon algoritması için gerekli olan ağırlıkları (-1 1 arasında) tutar. Ağırlıkların sabit omamasının sebebi eğer sabit değer olsaydı öğrenme sürecinde farklı özellikler arasında bir fark yaratılmazdı. Bu durum, özellikle sinir ağlarında, simetri problemi olarak bilinir ve öğrenmeyi engellerdi.

M V:Adam algoritması için gerekli olan hareketli ortalama ve kare ortalama.

```c
double **allocate_2d_array(int rows, int cols) {
    double **array = malloc(rows * sizeof(double *));
    if (!array) {
        perror("Bellek ayirma hatasi (satirlar)");
        exit(EXIT_FAILURE);
    }

    for (int i = 0; i < rows; i++) {
        array[i] = malloc(cols * sizeof(double));
        if (!array[i]) {
            perror("Bellek ayirma hatasi (sutunlar)");
            exit(EXIT_FAILURE);
        }
    }
    return array;
}

double activation(double z) {
    return tanh(z);
}

double activation_derivative(double z) {
    return 1.0 - tanh(z) * tanh(z);
}
```

Allocate_2d_array: Bellek üzerinde 2D bir dizi tahsis eder. Satır ve sütun sayısı alır, her satır için ayrı ayrı bellek ayırır(dinamik bellek ayırımı)

Activation: Bu fonksiyon çıktıyı -1 ile 1 arasında sıkıştırır. En son tahmin buna göre yapılır.

Activation_derivative: Bu fonksiyon aktivasyon fonksiyonun türevini alır.

```c
// CSV dosyasını okuma
void load_data(const char *filename, double data[DATA_SIZE][PIXELS + 1], double labels[DATA_SIZE]) {
    FILE *file = fopen(filename, "r");
    if (!file) {
        perror("CSV dosyasi acilamadi");
        exit(EXIT_FAILURE);
    }
    char line[8192];
    fgets(line, sizeof(line), file); // İlk satırı atla (başlıklar)
    int row = 0;
    while (fgets(line, sizeof(line), file) && row < DATA_SIZE) {
        char *token = strtok(line, ",");
        int original_label = atoi(token);
        if (original_label == 3) {
            labels[row] = 1.0;
        } else if (original_label == 7) {
            labels[row] = -1.0;
        } else {
            continue;
        }
        for (int col = 0; col < PIXELS; col++) {
            token = strtok(NULL, ",");
            data[row][col] = atof(token) / 255.0;
        }
        data[row][PIXELS] = 1.0;
        row++;
    }
    fclose(file);
}
```

Load_data: cvs formatındaki dosyayı okur , data ve labels matrisine atar. Eğer etiket 3 ise 1.0 olarak, 7 ise -1.0 olarak ayarlanır. Diğer etiketler atlanır. Piksel değerlerini [0, 1] aralığına dönüştürmek için her değeri 255.0 ile böler. Son sütun tüm girdilere 1.0 olarak atanır (bias için).

```
void split_data(double **train_data, double **test_data, double *train_labels,
double *test_labels,double data[DATA_SIZE][PIXELS+1],double labels[DATA_SIZE]) {
    int i;
    for (i = 0; i < TRAIN_SIZE; i++) {
        for (int j = 0; j <= PIXELS; j++) {
            train_data[i][j] = data[i][j];
        }
        train_labels[i] = labels[i];
    }
    for (; i < DATA_SIZE; i++) {
        for (int j = 0; j <= PIXELS; j++) {
            test_data[i - TRAIN_SIZE][j] = data[i][j];
        }
        test_labels[i - TRAIN_SIZE] = labels[i];
    }
}
```

Split_data: Eğitim ve test kümelerini oluşturur. İlk %80 eğitim, kalan %20 test verisi olarak ayrılır.

```
void initialize_weights(double *weights) {  //Eğer tüm ağ
    for (int i = 0; i <= PIXELS; i++) {
        weights[i] = ((double)rand() / RAND_MAX) * 2 - 1;
    }

}
```

Initialize_weights: Ağırlıkları rastgele küçük (-1 ve 1 arası)değerlerle başlatır (simetri kırılması için).

```
double forward(double *weights, double *x) {
    double sum = 0.0;
    for (int i = 0; i <= PIXELS; i++) {
        sum += weights[i] * x[i];
    }
    return activation(sum);
}
```

Forward: Ağırlıkların ve girdilerin çarpımını hesaplar (z = w * x) ve aktivasyon fonksiyonuna uygular.

```c
void gradient_descent_update(double *weights, double *x, double y) {
    double output = forward(weights, x);
    double error = y - output;
    double gradient = error * activation_derivative(output);

    for (int i = 0; i <= PIXELS; i++) {
        weights[i] += LEARNING_RATE * gradient * x[i];
    }
}

// SGD güncellemesi
void stochastic_gradient_descent_update(double *weights, double **data, double *labels, int data_size) {
    int random_index = rand() % data_size;
    double *x = data[random_index];
    double y = labels[random_index];
    gradient_descent_update(weights, x, y);
}
```

Gradient_descent_update: Tüm veri üzerinden gradient descent uygular. Hata (y - output) üzerinden gradyanı hesaplar ve ağırlıkları günceller.

Stochastic_gradient_descent_update: Rastgele bir veri seçer (mini-batch yerine tek örnekle çalışır) ve GD güncellemesini uygular.

```c
void adam_update(double *weights, double *x, double y) {
    t += 1; // Zaman adımını artır
    double output = forward(weights, x);
    double error = y - output;
    double gradient = error * activation_derivative(output);

    for (int i = 0; i <= PIXELS; i++) {
        m[i] = BETA1 * m[i] + (1 - BETA1) * gradient * x[i];
        v[i] = BETA2 * v[i] + (1 - BETA2) * gradient * x[i] * gradient * x[i];

        double m_hat = m[i] / (1 - pow(BETA1, t));
        double v_hat = v[i] / (1 - pow(BETA2, t));

        weights[i] += LEARNING_RATE * m_hat / (sqrt(v_hat) + EPSILON);
    }
}
```

Adam_update: Adam optimizasyon algoritmasını uygular. Hareketli ortalama (m) ve kare ortalama (v) hesaplar. Bias düzeltmeleri (m_hat, v_hat) ile ağırlıkları günceller.

```c
// Eğitim döngüsü
void train_and_log(double **train_data, double *train_labels, double *weights,
                   void (*update_fn)(double *, double *, double), const char *log_filename) {
    FILE *log_file = fopen(log_filename, "w");
    if (!log_file) {
        perror("Log dosyasi yazma hatasi");
        exit(EXIT_FAILURE);
    }
    fprintf(log_file, "Epoch,Loss\n");
    for (int epoch = 0; epoch < EPOC
        double loss = 0.0;
        for (int i = 0; i < TRAIN_SI
            double output = forward(weights, train_data[i]);
            double error = train_labels[i] - output;
            loss += error * error;
            update_fn(weights, train_data[i], train_labels[i]); // Ağırlıkları güncelle
        }
        loss /= TRAIN_SIZE;
        fprintf(log_file, "%d,%.4f\n", epoch + 1, loss);
        printf("Epoch %d/%d, Loss: %.4f\n", epoch + 1, EPOCHS, loss);
        if (loss < 0.0001) {
            break;
        }
    }
    fclose(log_file);
}
```

double *weights

Eğitim döngüsü

Train_and_log: Belirtilen optimizasyon algoritmasını kullanarak modeli eğitir. Kayıp (Loss) değerini dosyaya yazar ve ekrana bastırır. Ekstra olarak burda epoch loss grafiği için dosyaya yazma işlemleri bulunur.

Train_sgd_and_log: SGD için özel bir eğitim döngüsü içerir (her epoch için tüm veriler üzerinde SGD uygular).

```c
void evaluate(double **test_data, double *test_labels, double *weights) {
    int correct = 0;
    for (int i = 0; i < TEST_SIZE; i++) {
        double output = forward(weights, test_data[i]);
        int prediction = (output >= 0.0) ? 1 : -1;
        if (prediction == (int)test_labels[i]) {
            correct++;
        }
    }
    printf("Test Dogrulugu: %.2f%%\n", (correct / (double)TEST_SIZE) * 100);
}
```

Evaluate: Test verisi üzerinde modelin doğruluğunu hesaplar. Her test örneği için tahmin yapar ve gerçek etiketle karşılaştırır.

```c
void write_results_to_csv(const char *filename, double **test_data, double *test_labels, double *weights) {
    FILE *file = fopen(filename, "w");
    if (!file) {
        perror("CSV dosyasi yazma hatasi");
        exit(EXIT_FAILURE);
    }
    fprintf(file, "Gercek Etiket,Tahmin\n");
    int correct = 0;
    for (int i = 0; i < TEST_SIZE; i++) {
        double output = forward(weights, test_data[i]);
        int prediction = (output >= 0.0) ? 1 : -1;
        if (prediction == (int)test_labels[i]) {
            correct++;
        }
        fprintf(file, "%.0f,%d\n", test_labels[i], prediction);
    }
    double accuracy = (correct / (double)TEST_SIZE) * 100;
    fprintf(file, "\nDoğruluk: %.2f%%\n", accuracy);

    fclose(file);
}
```

write_results_to_csv: Test sonuçlarını ve genel doğruluğu bir CSV dosyasına yazar.

```c
// Veri kümeleri için bellek ayırma
double **train_data1 = allocate_2d_array(TRAIN_SIZE, PIXELS + 1);
double **test_data1 = allocate_2d_array(TEST_SIZE, PIXELS + 1);
double *train_labels1 = malloc(TRAIN_SIZE * sizeof(double));
double *test_labels1 = malloc(TEST_SIZE * sizeof(double));
load_data("filtered_data.csv", data1, labels1);

split_data(train_data1, test_data1, train_labels1, test_labels1, data1, labels1);
```

Main fonksiyon:

Dinamik bellek ayırma.

CSV dosyasından veri okunur ve eğitim/test kümelerine ayrılır.

```c
printf("Gradient Descent\n");
srand(time(NULL));
initialize_weights(weights_gd);
train_and_log(train_data1, train_labels1, weights_gd, gradient_descent_update, "loss_gd.csv");
evaluate(test_data1, test_labels1, weights_gd);

printf("\nStochastic Gradient Descent (SGD)\n");
srand(time(NULL));
initialize_weights(weights_sgd);
train_sgd_and_log(train_data1, train_labels1, weights_sgd, TRAIN_SIZE, "loss_sgd.csv");
evaluate(test_data1, test_labels1, weights_sgd);

printf("\nAdam Optimizasyonu\n");
srand(time(NULL));
initialize_weights(weights_adam);
train_and_log(train_data1, train_labels1, weights_adam, adam_update, "loss_adam.csv");
evaluate(test_data1, test_labels1, weights_adam);
```

Gradient Descent (GD):Ağırlıklar başlatılır.GD ile model eğitilir, sonuçlar kaydedilir ve test doğruluğu hesaplanır.

Stochastic Gradient Descent (SGD):Ağırlıklar başlatılır.SGD ile model eğitilir, sonuçlar kaydedilir ve test doğruluğu hesaplanır.

Adam Optimizasyonu:Ağırlıklar başlatılır.Adam ile model eğitilir, sonuçlar kaydedilir ve test doğruluğu hesaplanır.

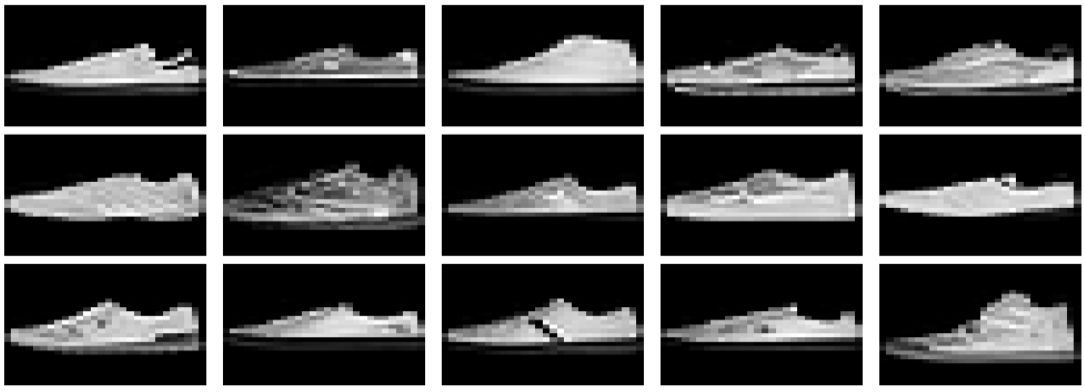
```
for (int i = 0; i < TRAIN_SIZE; i++)
    free(train_data1[i]);

for (int i = 0; i < TEST_SIZE; i++)
    free(test_data1[i]);

free(train_data1);
free(test_data1);
free(train_labels1);
free(test_labels1);
return 0;
```

Bellek serbest bırakılır.

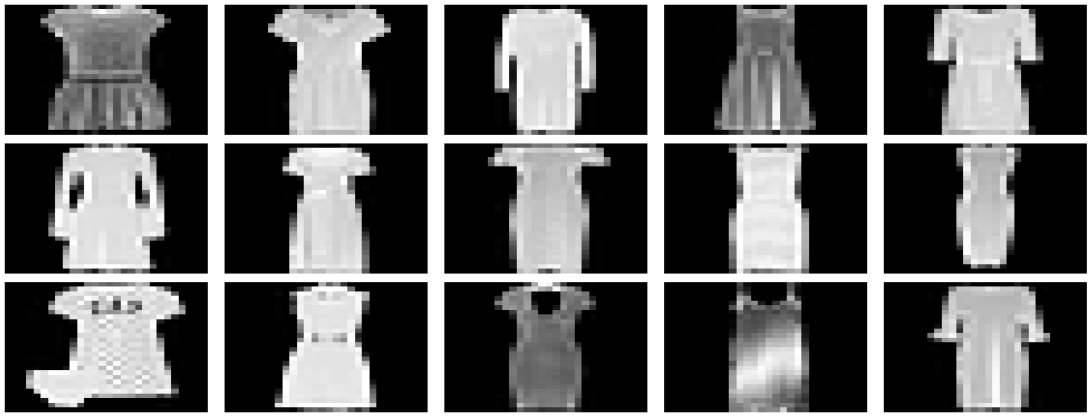
Veri seti

Veri seti olarak 28*28 pikseller içeren mnist_fashion kullanılır 1250 tane sneaker ve 1250 tane dress içerir

Sneakers datası:

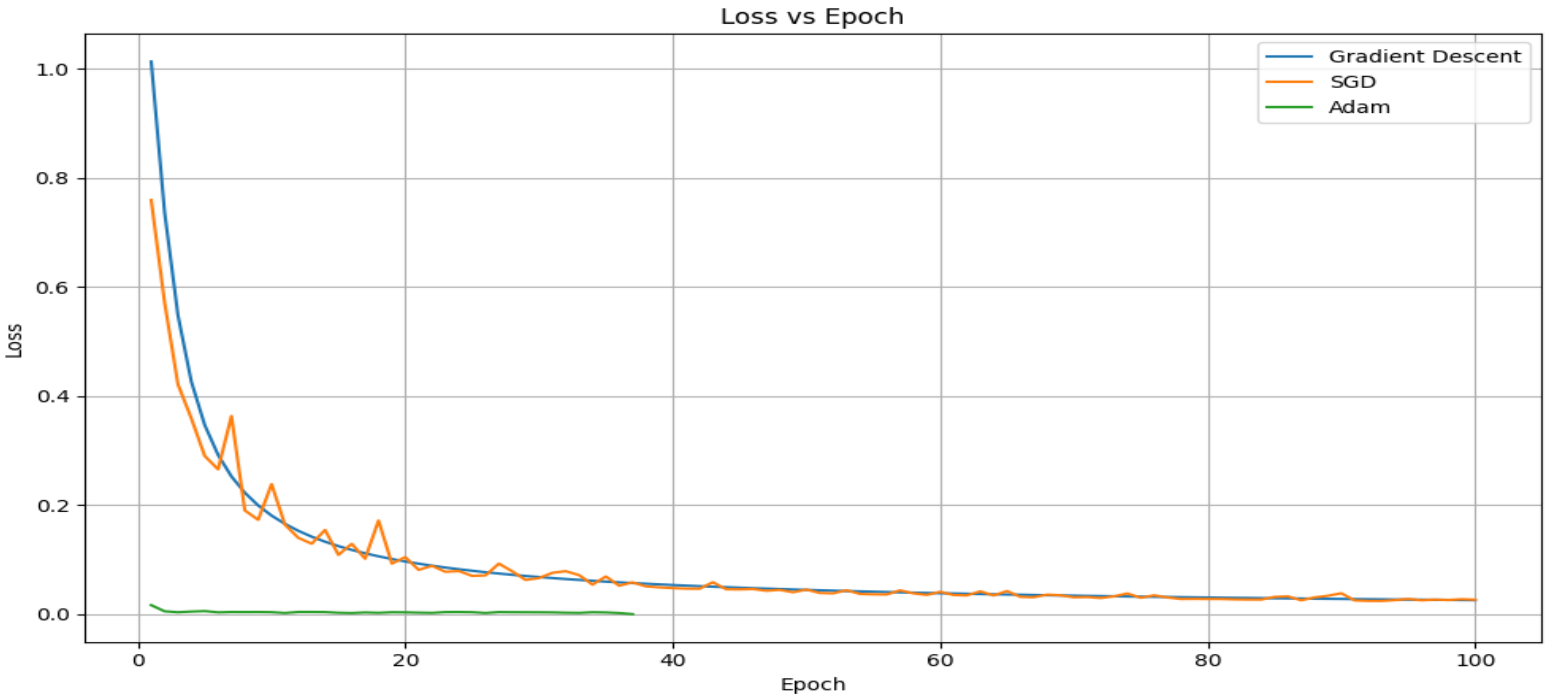


Dresses datası:

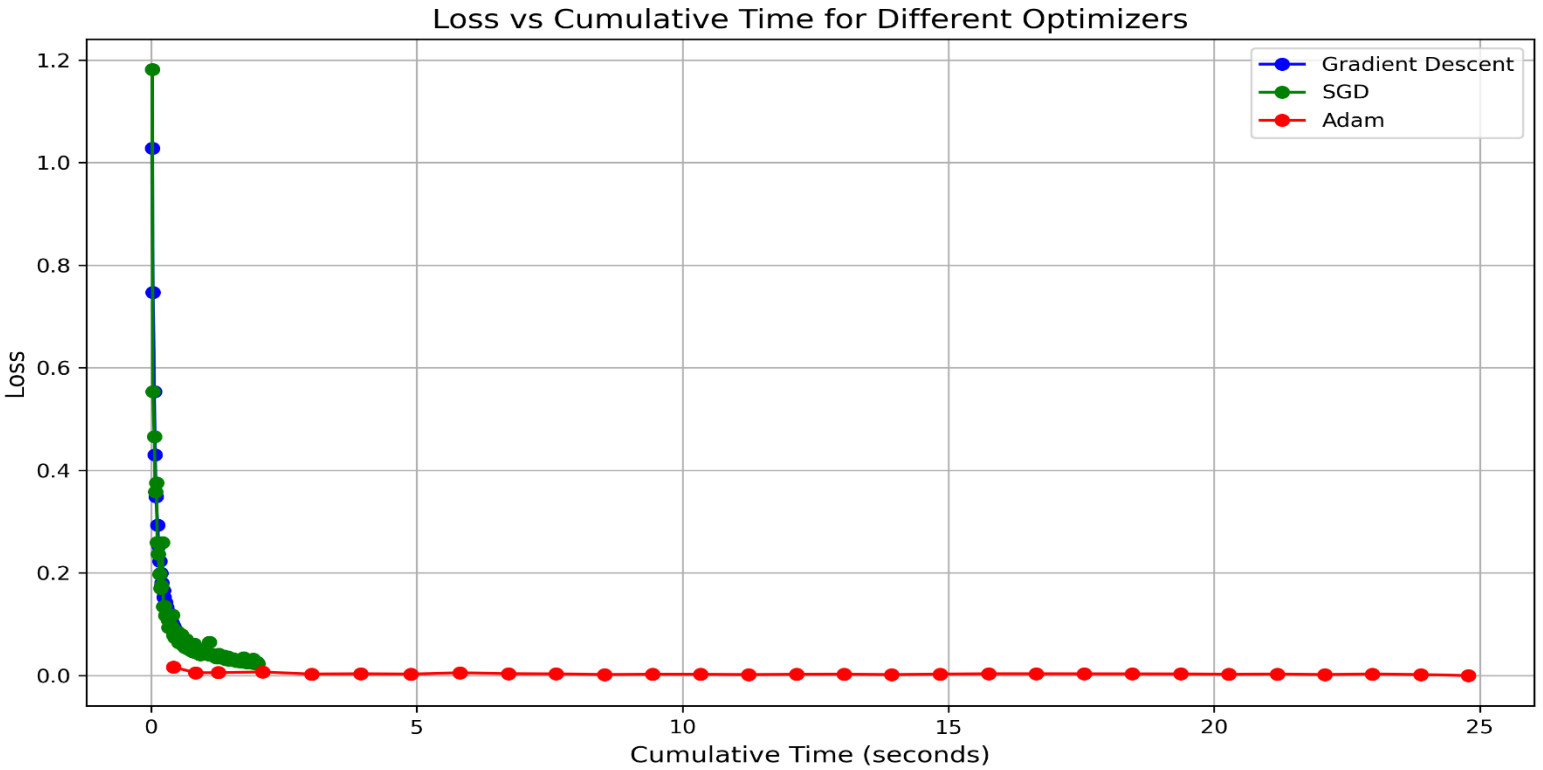


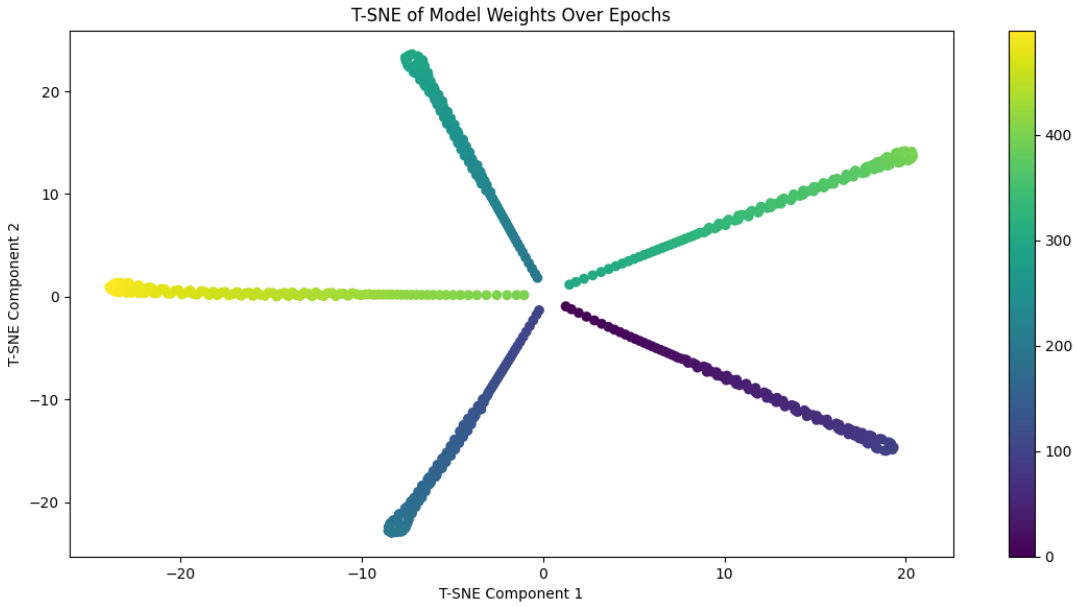
Grafikler:

Epoch and loss:

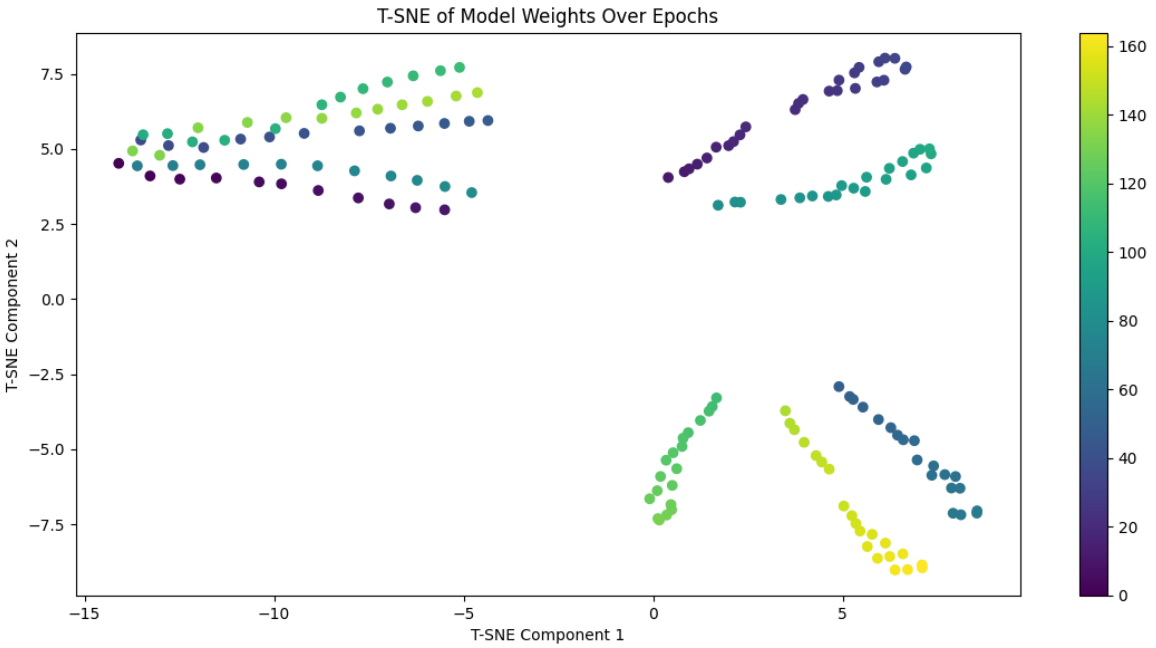


Loss and time:

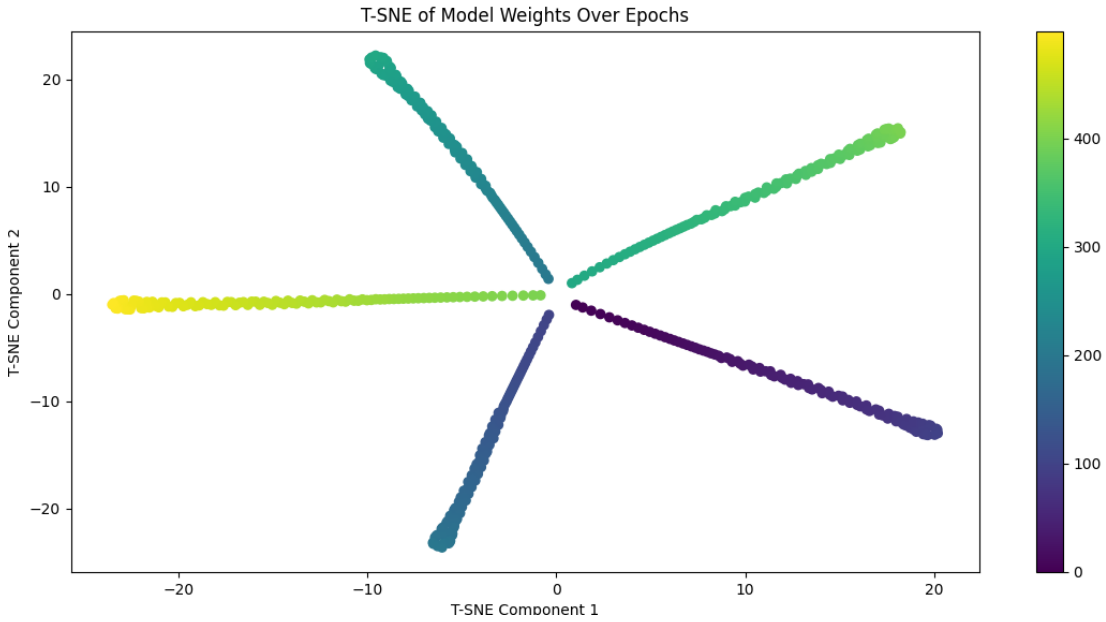




Bu grafik, bir modeli eğitirken **Stochastic Gradient Descent (SGD)** optimizasyon yönteminin farklı ağırlık uzaylarındaki zamanla evrimini t-SNE (t-Distributed Stochastic Neighbor Embedding) yöntemiyle görselleştiriyor. Renk skalası, modelin eğitim sürecindeki **epoch** sayısını ifade ediyor. Sarıya yakın noktalar eğitimin başındaki ağırlık durumlarını, mora yakın noktalar ise eğitimin sonundaki ağırlık durumlarını temsil ediyor. Renk skalası, modelin eğitim sürecindeki **epoch** sayısını ifade ediyor. Sarıya yakın noktalar eğitimin başındaki ağırlık durumlarını, mora yakın noktalar ise eğitimin sonundaki ağırlık durumlarını temsil ediyor. Eğer modelin parametre uzayı karmaşıksa ve eğitim için kullanılan veri gürültülü ise, t-SNE böyle ayrışmalar gösterebilir. Dalların net ayrılması, SGD'nin farklı ağırlık gruplarını optimize ettiğini ve modelin bu doğrultuda farklı karar bölgelerine oturduğunu gösterebilir.



Bu grafik, bir optimizasyon sürecinde **AdaGrad** gibi adaptif öğrenme oranı kullanan bir yöntemin ağırlık uzayındaki evrimini t-SNE ile görselleştiriyorYatay ve dikey eksendeki noktalar farklı epoch'lardaki model ağırlıklarını temsil ediyor. Renk skalası (mora yakın başlangıç, sarıya yakın bitiş) optimizasyonun zaman içindeki ilerlemesini gösteriyor. Grafik dört temel küme veya farklı bölgelere ayrışmış yapılar sergiliyor. Bu, ağırlık uzayında optimizasyonun belirli yönlere doğru net bir şekilde ilerlediğini ve farklı yerel minimumlara doğru yaklaşıldığını gösterebilir. Üstteki kümeler ve alttaki kümeler arasında bir mesafe var. Bu, parametrelerin birbirinden oldukça farklı bölgelere taşındığını işaret edebilir. AdaGrad gibi algoritmalar genellikle farklı ağırlık gruplarını farklı hızlarda optimize eder. Bu grafik, optimizasyonun sonunda belirli sabit noktalara (muhtemel minimumlara) ulaştığını gösteriyor. Ağırlıkların farklı kümelere ayrılması, modelin **değişkenlere göre özelleşmiş öğrenme yolları** geliştirdiğini işaret ediyor.



Bu grafik, **Gradient Descent (GD)** optimizasyon algoritmasının model ağırlıklarının t-SNE ile görselleştirilmiş evrimini göstermektedir. **Yıldız şeklinde bir dağılım** gözlemleniyor. Bu durum, model ağırlıklarının optimizasyon sürecinde farklı yönlerdeki gradyanlarla ilerlediğini ve **potansiyel yerel minimumlara** yaklaştığını gösterebilir. Gradient Descent'in doğası gereği, tüm parametreler aynı öğrenme oranıyla güncellendiği için, model parametrelerinin bu simetrik yapıyı sergilemesi beklenebilir. Bu grafik, Gradient Descent'in **sabit öğrenme oranı** ile çalıştığını ve tüm ağırlıkların paralel bir şekilde ilerlediğini ortaya koyuyor.

KODU AÇIKLAYAN VİDEO:
https://www.youtube.com/watch?v=qH5uKz3aNZU