Bilgisayar Mühendisliği İçin Diferansiyel Denklemler Ödevi

Ad: Bilal Efe

Soyad: Uysal

Öğrenci Numarası: 22011031

Video Adresi: https://youtu.be/thmZuTOwkHQ

Başlangıç

Öncelikle dataset'imiz olarak iki tane görsel sınıf olacak şekilde 100 tane 4 rakamı 100 tane ise 7 rakamını oluşturdum. Görseller aşağıdaki gibi oldu.

Sonra bu fotoğrafları .csv formatında bir excel dosyasına dönüştürdüm bu excel dosyası [200] satır [785] stündan oluşuyor. Görseller 28*28 ‘ lik görseller olduğundan her birini bir satıra ekledim ve son hücrelerine bias değeri olarak 1 değerini ekledim. Sonra ise bu .csv formatındaki dosyayı kod ortamına aldım ve bir matrise atadım.

```c
void read_csv(const char *filename, double **matrix) {
    FILE *file = fopen(filename, "r");
    if (!file) {
        printf("Dosya acilamadi: %s\n", filename);
        return;
    }

    char line[LINE_LENGTH];
    int row = 0;

    // Dosyayı satır satır oku
    while (fgets(line, sizeof(line), file) && row < 200) {
        int col = 0;
        char *token = strtok(line, ";");

        // Satırdaki her sütunu virgülle ayırarak okuma
        while (token && col < 785) {
            matrix[row][col] = atof(token);  // Veriyi sayıya çevirerek matrise ekle
            token = strtok(NULL, ";");
            col++;
        }
        row++;
    }

    fclose(file);
}
```

Bu kod parçası sayesinde dışardaki .csv dosyasını iki boyutlu x matrisine aktardım.

Sonra bu dataları tutan x matrisini eğitim ve test için ayırmam gerekti ve aşağıdaki fonksiyonla x matrisini 80 + 80 = 160 eğitim kümesi ve 20 + 20 = 40 tane de test kümesi olarak ayırdım.

```c
void separation_train_Test(double **x,double **x_train,double **x_test){
    int i,j;
    for(i=0; i<80; i++){
        for(j=0;j<785;j++){
            x_train[i][j] = x[i][j];
        }

    }
    for(i=100; i<180; i++){
        for(j=0;j<785;j++){
            x_train[i-20][j] = x[i][j];
        }

    }
    for(i=80; i<100; i++){
        for(j=0;j<785;j++){
            x_test[i-80][j] = x[i][j];
        }

    }
    for(i=180; i<200; i++){
        for(j=0;j<785;j++){
            x_test[i-160][j] = x[i][j];
        }

    }
}
```

Ve sonra bizim belirlediğimiz bir başlangıç değerine göre w matrisini aşağıdaki fonksiyon ile oluşturdum.

```
void generate_w(double *w,double baslangic){
    int i;
    for(i=0;i<785;i++){
        w[i] = baslangic;
    }

}
```

Sonrasında y diye tek boyutlu bir matris oluşturdum ve bu y matrisi 4 resimleri için 1 değerini tutup 7 resimleri için -1 değerini tutacak şekilde 200 veri için düzenledim.

Gradient Descent'in Oluşturulması

```
void gradient_descent(double **x,double **x_train,double **x_test,double *w,double **w_history,int *y,double loss_history[101],doub
    int i,j,iteration;
    double sum,accuracy_rate,duration;
    int result[40];
    clock_t start,end;


    for(i=0;i<785;i++){
        w_history[0][785] = w[i];
    }

    loss_history[0] = calculate_loss(w,x_train,y);
    time_history[0] = 0;
    int flag = 0;
    for(iteration = 1;iteration <101;iteration++){
        start = clock();
        gradient_descent_w_edit(w,x_train,y);
        w_history_edit(w,w_history,iteration);

        loss_history[iteration] = calculate_loss(w,x_train,y);
        end = clock();

        duration = ((double)(end - start)) / CLOCKS_PER_SEC;

        if(flag == 0){
            time_history[1] = duration;
            flag = 1;
        }else{
            time_history[iteration] = time_history[iteration-1] + duration;
        }

    }
```

Verileri eğitim aşamasında ilk başta w değerinin başlangıç değerlerini w_history dizisinin ilk satırına ekledim.

Sonra her loss değerlerini grafik oluşturabilemek için her iterasyonda aşağıdaki kod parçası sayesinde loss değerlerini tutuyorum.

```c
double calculate_loss(double *w,double **x_train,int *y){
    double sum, sum_loss = 0;
    int i,j;

    for(i=0;i<160;i++){
        sum = 0;
        for(j=0;j<785;j++){
            sum += w[j]*x_train[i][j];
        }

        if(i<80){
            sum_loss += pow((y[i]-tanh(sum)),2);
        }else{
            sum_loss += pow((y[i+20]-tanh(sum)),2);
        }
    }

    return (sum_loss / 160.0);


}
```

Sonra ise w değerlerini GD yöntemine göre aşağıdaki fonksiyon ile düzenledim.

```c
void gradient_descent_w_edit(double *w,double **x_train,int *y ){
    int i,j;
    double sum;


    for(j=0;j<785;j++){

        sum = 0;

        for(i=0;i<160;i++){

            if(i<80){
                sum += derivative_loss(w[j],x_train[i][j],y[i]);
            }else{
                sum += derivative_loss(w[j],x_train[i][j],y[i+20]);
            }

        }

        w[j] = w[j] - EPS*(sum/160.0) ;



    }

}
```

Bu fonksiyonda ayrıyeten derivative_loss fonksiyonu ile loss fonksiyonunun türevini hesapladım. Aşağıda da o fonksiyonun kodu var.

```
double derivative_loss(double w,double x,int y){

    return -2*(y-tanh(w*x))*(1-pow(tanh(w*x),2))*x;


}
```

Ana GD kod parçasında w_history_edit diye bir fonksiyon kullandım bu fonksiyon ise o anki w değerlerini alıp ilerde grafik oluşturmada kullanmamız için w_history dizisine atıyor. Bu bölümün kod parçası aşağıda.

```
void w_history_edit(double *w, double **w_history, int iteration ){
    int i;
    for(i=0;i<785;i++){
        w_history[iteration][i] = w[i];
    }
}
```

Ana kod parçasında göründüğü gibi her iterasyonun başında ve sonunda o anki zaman değerini aldım ve bu değerleri çıkartarak o iterasyonda geçen zamanı buldum. Bu geçen zamanları ise her iterasyonda time_history dizisine aktardım.

Stochastic Gradient Descent'in Oluşturulması

```
void stochastic_gradient(double **x,double **x_train,double **x_test, double *w, double **w_history,int *y,double
    int i,j,iteration,result[40],flag = 0;
    double sum,accuracy_rate,duration;
    clock_t start,end;

    for(i=0;i<785;i++){
        w_history[0][785] = w[i];
    }

    loss_history[0] = calculate_loss(w,x_train,y);
    time_history[0] = 0;

    for(iteration = 1;iteration <101;iteration++){
        start = clock();
        stochastic_gradiant_w_edit(w,x_train,y);
        w_history_edit(w,w_history,iteration);

        loss_history[iteration] = calculate_loss(w,x_train,y);
        end = clock();

        duration = ((double)(end - start)) / CLOCKS_PER_SEC;

        if(flag == 0){
            time_history[1] = duration;
            flag = 1;
        }else{
            time_history[iteration] = time_history[iteration-1] + duration;
        }

    }
```

Burada G
algoritm
ım.

```
void stochastic_gradiant_w_edit(double *w,double **x_train,int *y){
    int i,j;
    double derivative;

    i = rand() % 160;

    for(j=0;j<785;j++){
        if(i<80){
            derivative = derivative_loss(w[j],x_train[i][j],y[i]);
        }else{
            derivative = derivative_loss(w[j],x_train[i][j],y[i+20]);
        }
        w[j] = w[j] - EPS*(derivative);
    }


}
```

Geri kalan düzenlemeler ve algoritma kısımları GD ile aynı işlemler.

ADAM Algoritmasının Düzenlenmesi

```
void Adam_Algorithm(double **x,double *w,double **x_train,double **x_test,double **w_history,int *y,double loss_history[101],double time_history[101]){
    double learningRate = 0.001,b1 = 0.9,b2 = 0.999,e = pow(10,-8),gt,mt=0,vt=0,sum,accuracy_rate,duration;
    int i,j,iteration,result[40],flag=0;
    clock_t start,end;

    for(i=0;i<785;i++){
        w_history[0][785] = w[i];
    }
    loss_history[0] = calculate_loss(w,x_train,y);
    time_history[0] = 0;
    for(iteration = 1;iteration<101;iteration++){
        start = clock();

        j = rand() % 160;

        for(i=0;i<785;i++){
            if(j<80){
                gt = derivative_loss(w[i],x_train[j][i],y[j]);
            }else{
                gt = derivative_loss(w[i],x_train[j][i],y[j+20]);
            }

            mt = (b1 * mt) + (1.0 - b1)*gt;
            vt = (b2 * vt) + (1.0 - b2)*pow(gt,2);

            w[i] -= (learningRate*mt)/sqrt(vt + e);

        }

        w_history_edit(w,w_history,iteration);

        loss_history[iteration] = calculate_loss(w,x_train,y);

        end = clock();

        duration = ((double)(end - start)) / CLOCKS_PER_SEC;

        if(iteration == 1){
            time_history[1] = duration;
            flag = 1;
        }else{
            time_history[iteration] = time_history[iteration-1] + duration;
        }
```

Burada anahatlarıyla aslında yaptığım bazı işlemler GD ve SGD ile aynı şeyler farklı olan ise w güncellemeleri bu güncellemeleri aşağıdaki görseldeki algoritmanın kodunu yazarak yaptım.

$t \leftarrow t + 1$
$g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$ (Get gradients w.r.t. stochastic objective at timestep t)
$m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ (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)

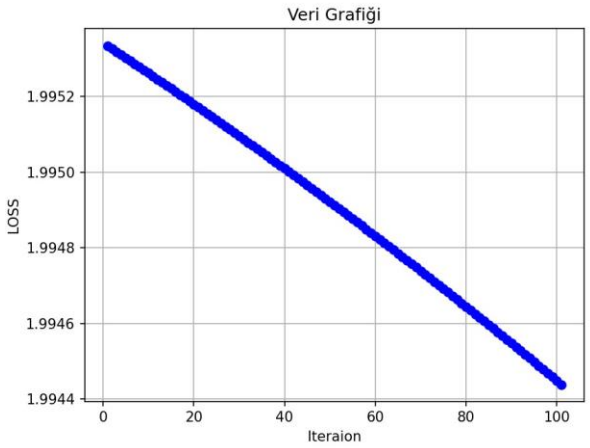
Ve geri kalan zaman hesaplamaları, eski w değerlerini diziye aktarma, türev alma, loss hesaplama bunların hepsi diğer algoritmalarla aynı.

Test aşaması hepsi için aynıydı burada test kümesindeki dataları yaptığımız optmizasyonda test edip eğer 1 sınıfı için çıktı 0.5 ten büyükse doğru bulduğunu değilse yanlış bulduğunu bir matriste tuttum. Eğer -1 sınıfı için değer -0.5 ten küçükse bu değerin doğru olduğunu tuttum. Ve en sonda doğru bulduklarımın sayısını toplam teste bölerek bu eğitimin başarı oranını hesapladım.

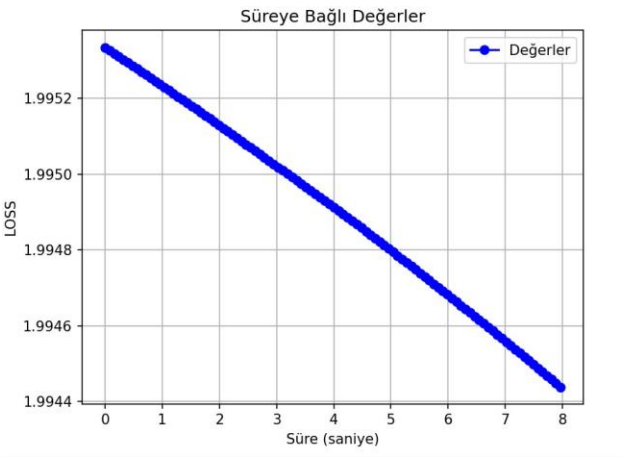
```
for(i=0;i<40;i++){
    sum =0;
    for(j=0;j<785;j++){
        sum += x_test[i][j]*w[j];
    }

    if(i<20){

        if(tanh(sum) >= 0.5){
            result[i] = 1;
        }else{
            result[i] = 0;
        }


    }else{
        if(tanh(sum) <= -0.5){
            result[i] = 1;
        }else{
            result[i] = 0;
        }
    }
}

sum = 0;

for(i=0;i<40;i++){
    sum += result[i];
}

accuracy_rate = sum/40.0;
```

TIME-LOSS Ve ITERATION-LOSS Grafikleri

GD:

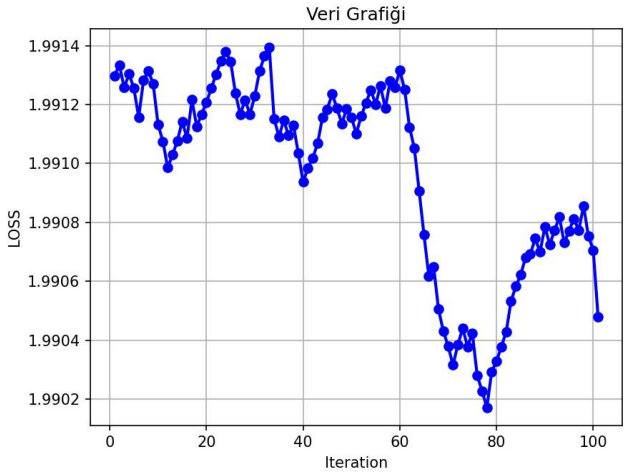


ITERATON-LOSS

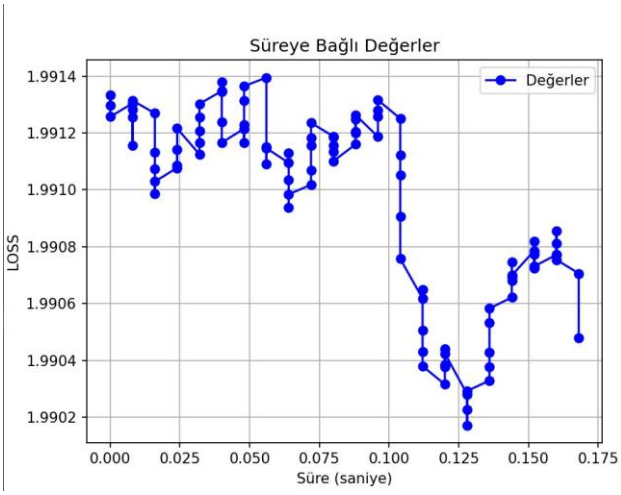


TİME-LOSS

SGD:

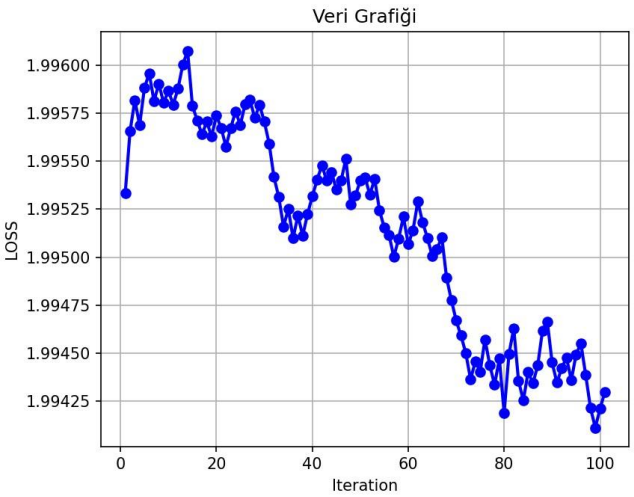


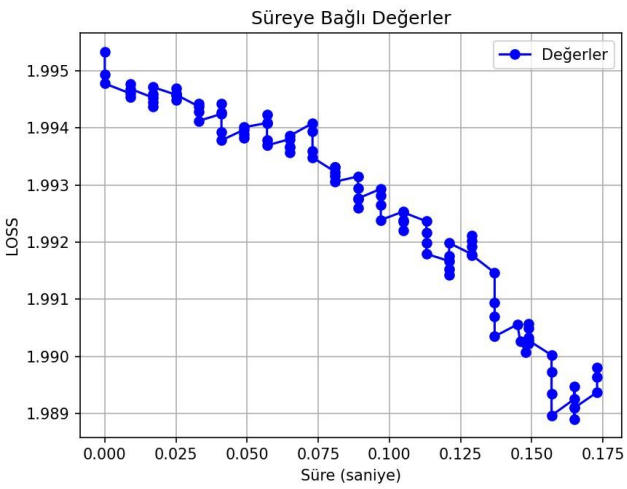
ITERATION-LOSS



TIME-LOSS

ADAM:



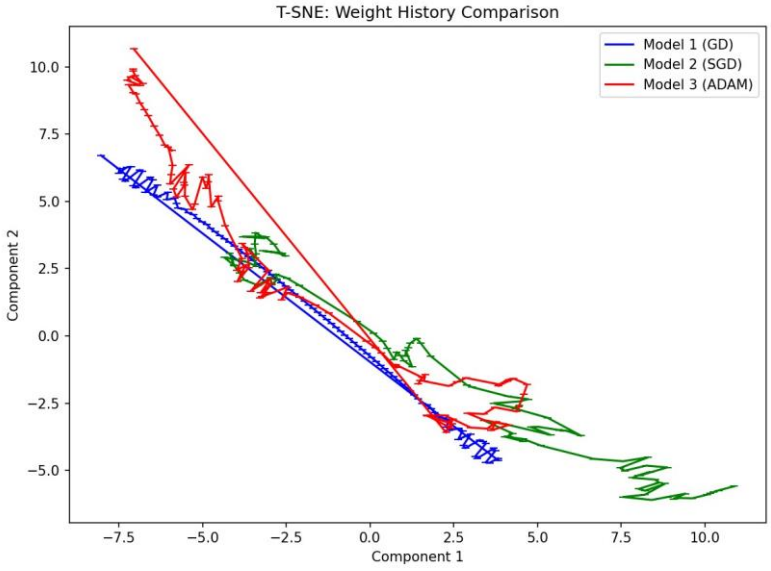


ITERATION-LOSS **TİME-LOSS**

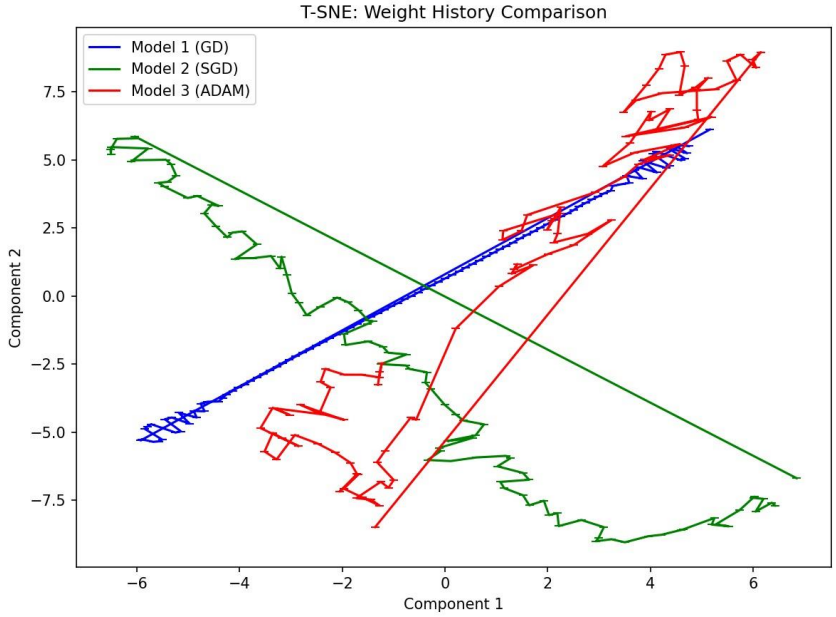
GD algoritması, SGD ve ADAM 'a göre daha yavaş çalışıyor ama daha az gürültülü şekilde ilerliyor. SGD 'de gürültünün çok olma sebebi her iterasyonda datayı rastgele seçmesinden kaynaklanıyor.

Farklı W Başlangıç değerlerine göre $W_{1:t}$ Grafikleri

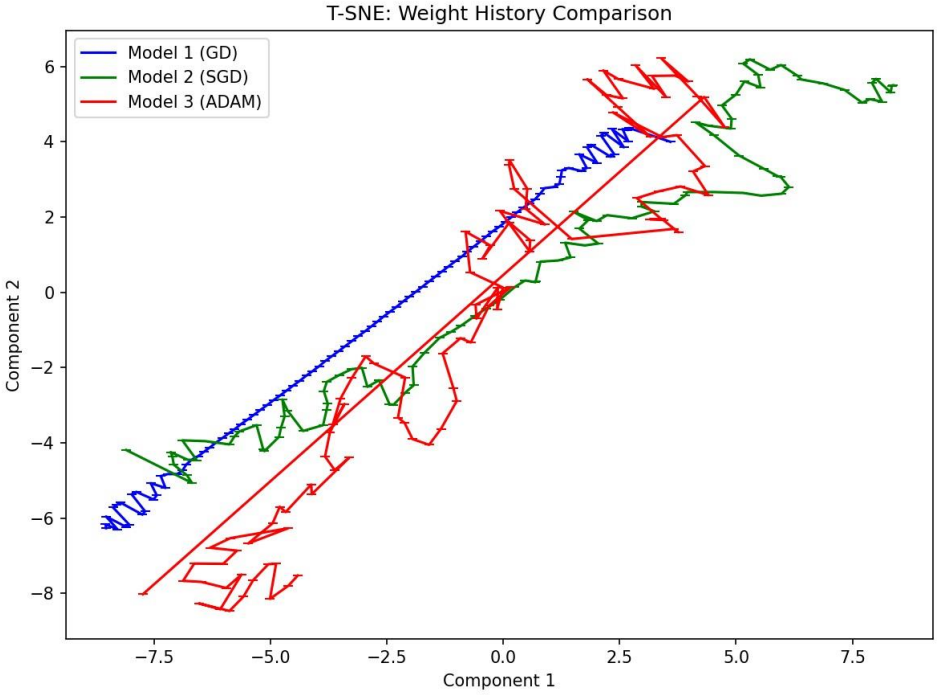
W = 0.1:



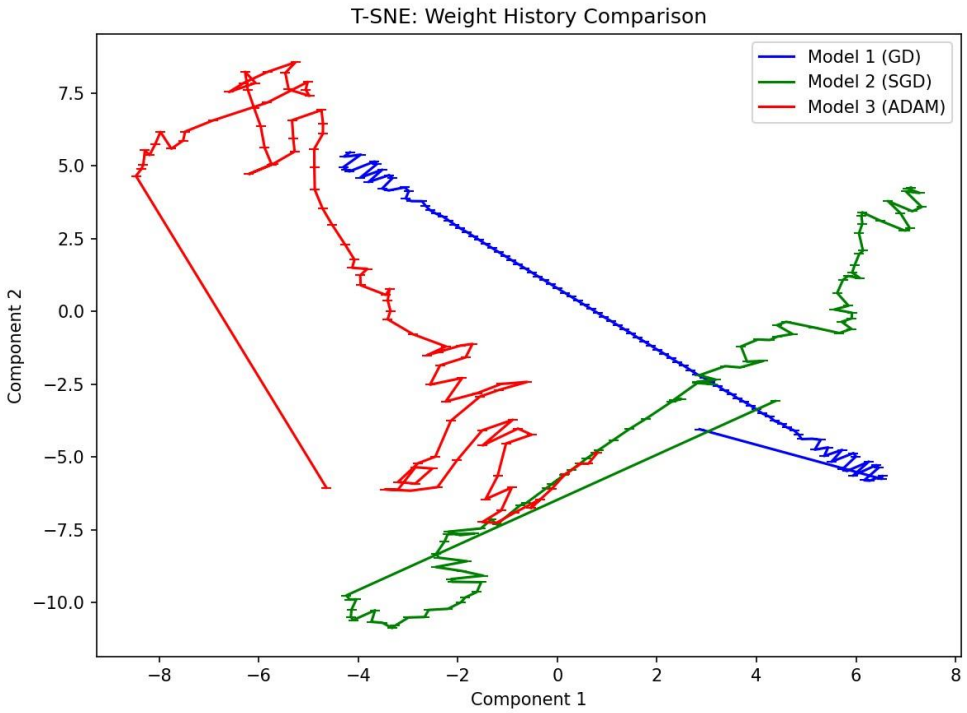
W = -0.1:


T-SNE: Weight History Comparison
Model 1 (GD)
Model 2 (SGD)
Model 3 (ADAM)
Component 2
Component 1
7.5
5.0
2.5
0.0
−2.5
−5.0
−7.5
−6
−4
−2
0
2
4
6

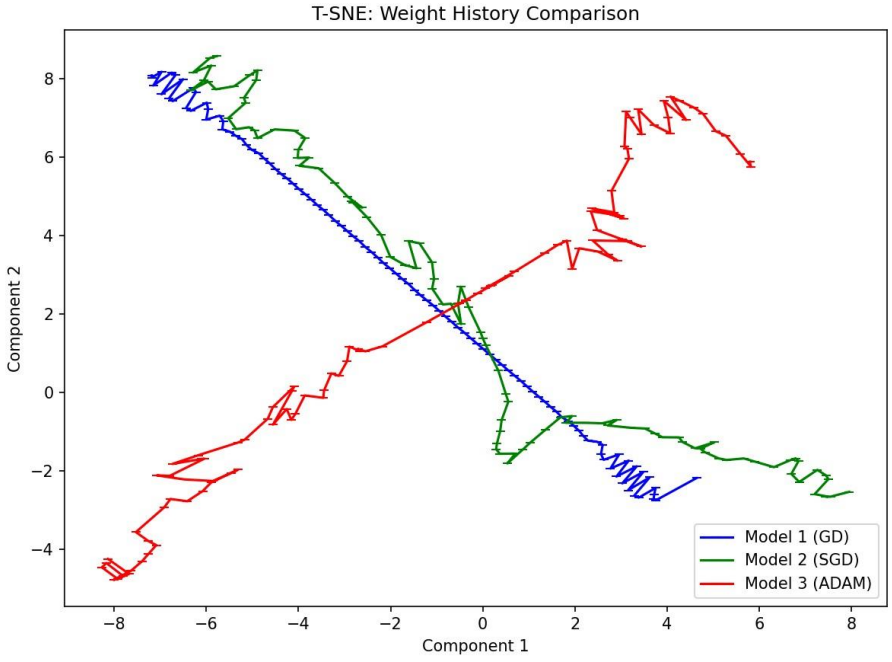
W = 0.01:


T-SNE: Weight History Comparison
Model 1 (GD)
Model 2 (SGD)
Model 3 (ADAM)
Component 2
Component 1
6
4
2
0
−2
−4
−6
−8
−7.5
−5.0
−2.5
0.0
2.5
5.0
7.5

W = -0.01:



W = 0.001:



GD daha düz şekilde ilerlerken diğerleri dalgalı ve zik-zak şeklinde ilerliyor.