KARADENİZ TEKNİK ÜNİVERSİTESİ
MÜHENDİSLİK FAKÜLTESİ
BİLGİSAYAR MÜHENDİSLİĞİ BÖLÜMÜ

BIL4015 YAPAY SİNİR AĞLARI DERSİ
DÖNEM PROJESİ

Adı Soyadı
Osman Can AKSOY – 394797

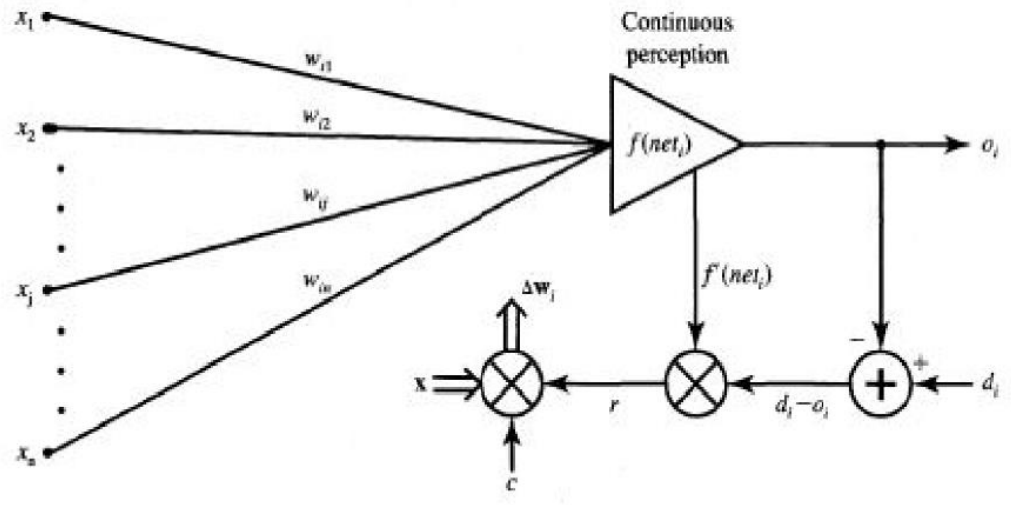
Dersin Sorumlusu
Prof. Murat EKİNCİ

2023-2024 GÜZ DÖNEMİ

Ödev 1: Tek katmanlı yapay sinir ağı ile :

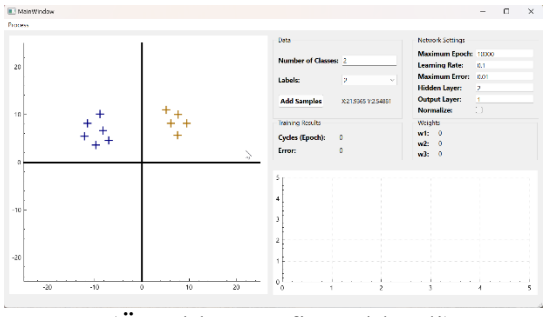
1.1. İki sınıfa ait 2-Boyutlu veri uzayında etiketli örneklerden :
sürekli aktivasyon fonksiyonlu danışmanlı öğrenme (supervised learning), test ve güncelleme süreçleri

Sürekli Aktivasyon Fonksiyonlu(Delta) Öğrenme Kuralı:

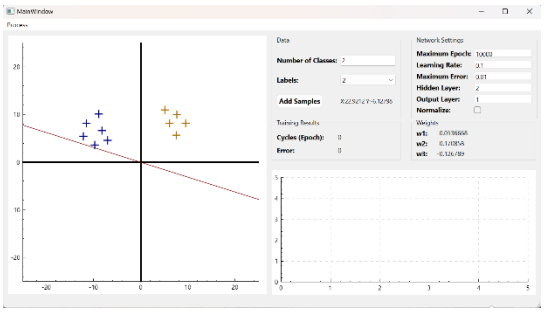


Delta kuralı genellikle ağırlık güncellemesi için kullanılır. Temelde, ağın tahminleri ile gerçek veri arasındaki hatayı ölçer ve bu hatayı azaltmak için ağdaki ağırlıkları günceller. Bu güncelleme, ağırlıkları belirli bir öğrenme oranı ile hatanın bir fonksiyonu olan delta ile çarparak yapılır.

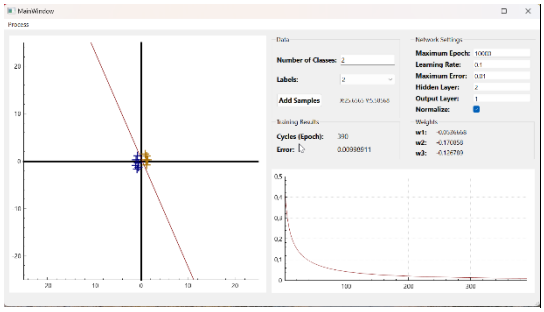
1.1 Uygulama:



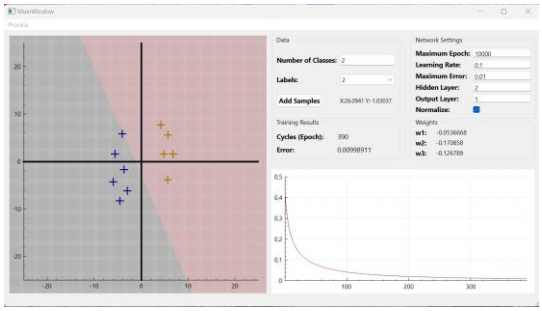
(Örnekler Grafiğe Eklendi)



(Ağırlıklar Rastgele Atandı)



(Eğitim Normalizasyon ile Yapıldı)



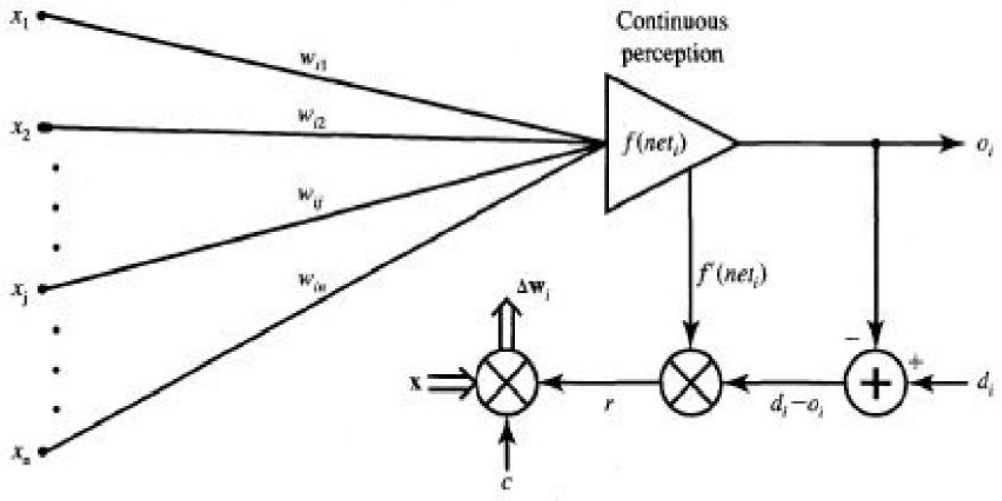
(Test İşlemi Yapıldı)

1.1 Kaynak Kodları:

```
float single_neuron_delta_rule(float *inputs, float *weights, float *targets, float* bias, float lr, int sample_size, int input_dim)
{
    float error = 0.0;
    for(int i = 0; i < sample_size; i++) {
        float net = 0;
        for(int j = 0; j < input_dim; j++) {
            net += weights[j]*inputs[i * input_dim + j];
        }
        net += bias[0];

        float output = sigmoid(net);
        float output_der = sigmoid_derivative(net);

        int desired;
        if(targets[i] == 1) {
            desired = -1;
        }
        else {
            desired = 1;
        }

        for(int j = 0; j < input_dim; j++) {
            weights[i] += lr * (desired - output) * output_der *inputs[i * input_dim + j];
        }
        bias[0] += lr * (desired - output) * output_der * 1;

        error += 0.5 * (desired - output) * (desired - output);
    }
    error /= sample_size;
    return error;
}
```

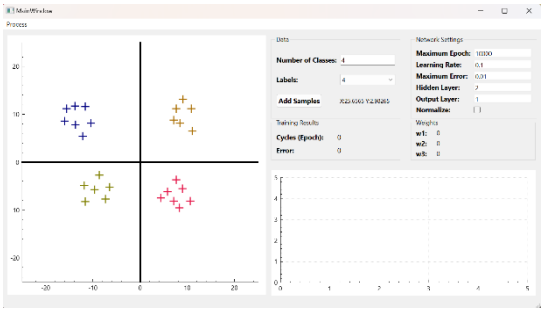
1.2. Çoklu sınıflara ait ait 2-Boyutlu veri uzayında etiketli örneklerden : sürekli aktivasyon fonksiyonlu danışmanlı öğrenme (supervised learning), test ve güncelleme süreçleri

Sürekli Aktivasyon Fonksiyonlu(Delta) Öğrenme Kuralı:

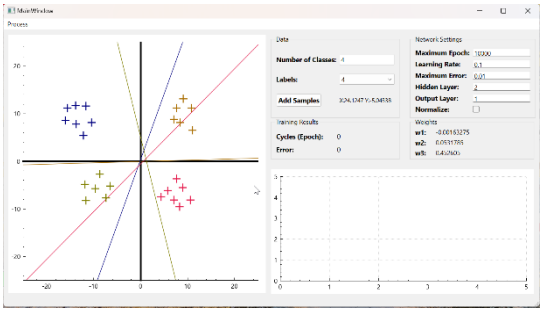


Delta kuralı genellikle ağırlık güncellemesi için kullanılır. Temelde, ağın tahminleri ile gerçek veri arasındaki hatayı ölçer ve bu hatayı azaltmak için ağdaki ağırlıkları günceller. Bu güncelleme, ağırlıkları belirli bir öğrenme oranı ile hatanın bir fonksiyonu olan delta ile çarparak yapılır.

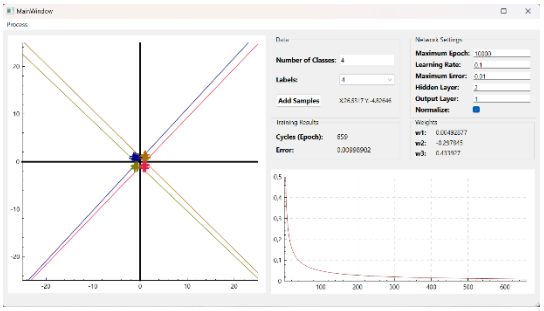
1.2 Uygulama:



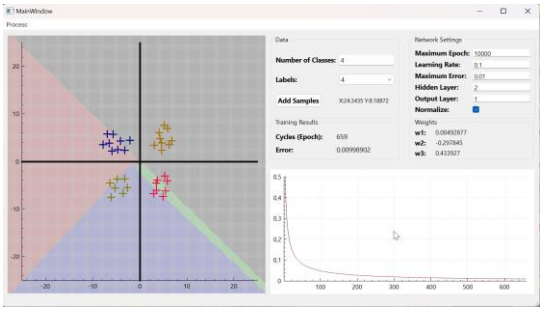
(Örnekler Grafiğe Eklendi)



(Ağırlıklar Rastgele Atandı)



(Eğitim Normalizasyon ile Yapıldı)



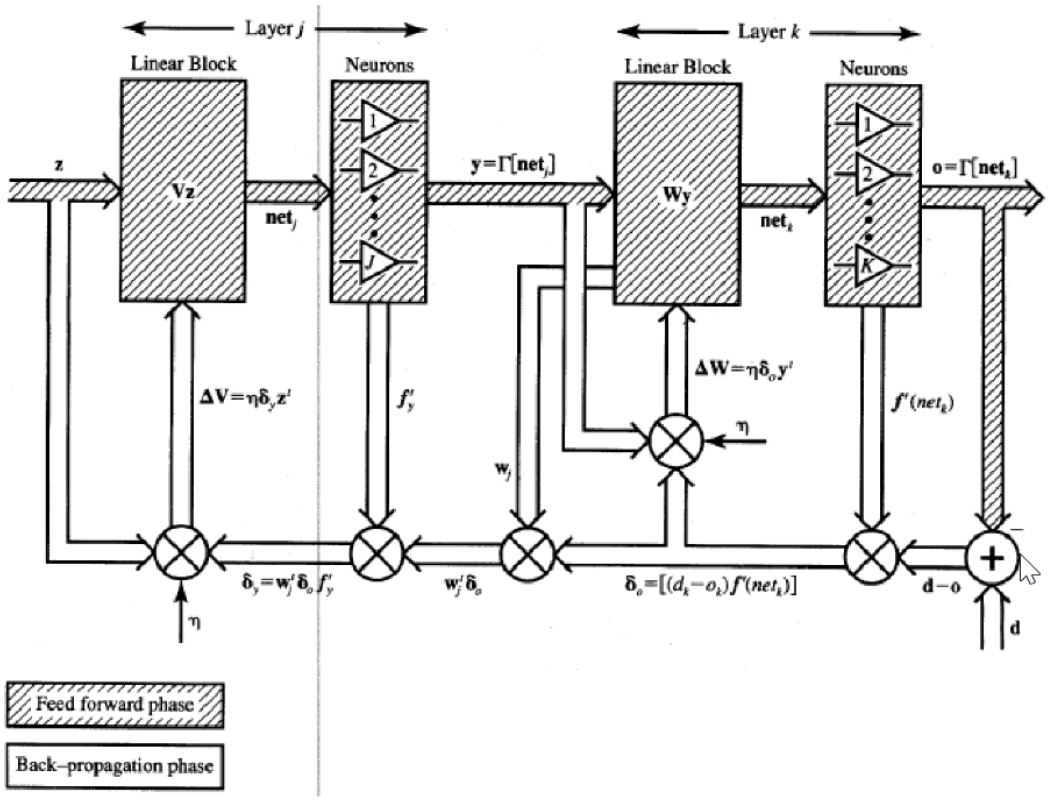
(Test İşlemi Yapıldı)

1.2 Kaynak Kodları:

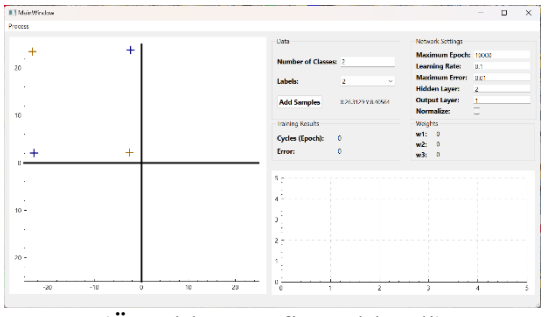
```
float multi_class_delta_rule(float *inputs, float *weights, std::vector<float> targets, float* bias, float lr, int sample_size, int input_dim, int class_count)
{
    float total_error = 0.0;

    for(int i = 0; i < sample_size; i++) {
        std::vector<float> net(class_count, 0.0);
        for(int j = 0; j < input_dim; j++) {
            for (int classIndex = 0; classIndex < class_count; classIndex++)
            {
                net[classIndex] += weights[classIndex * input_dim + j] * inputs[i * input_dim + j];
            }
        }

        for (int classIndex = 0; classIndex < class_count; classIndex++)
        {
            net[classIndex] += bias[classIndex];
        }

        for (int classIndex = 0; classIndex < class_count; classIndex++)
        {
            float output = sigmoid(net[classIndex]);
            float output_der = sigmoid_derivative(net[classIndex]);

            float target = targets[i * class_count + classIndex];

            total_error += 0.5 * (target - output) * (target - output);

            for (int k = 0; k < input_dim; k++)
            {
                weights[classIndex * input_dim + k] += lr * (target - output) * output_der * inputs[i * input_dim + k];
            }

            bias[classIndex] += lr * (target - output) * output_der;
        }
    }

    total_error /= sample_size;
    return total_error;
}
```

Ödev 2: Çok katmanlı yapay sinir ağı:

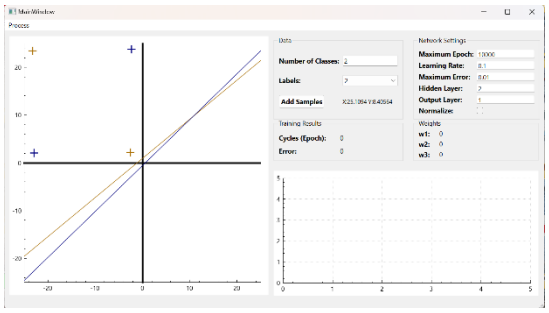
2.1 Çoklu sınıflara ait 2-Boyutlu veri uzayında etkiketli örneklerden hatanın geri yayılımı (error-back-propagation) tabanlı SGD, Momentum işlevli danışmanlı öğrenme (supervised learning).



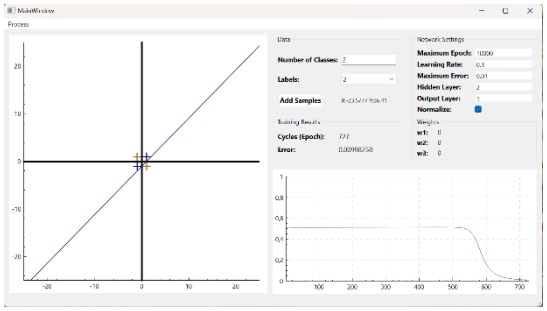
2.1 Uygulama(Delta):



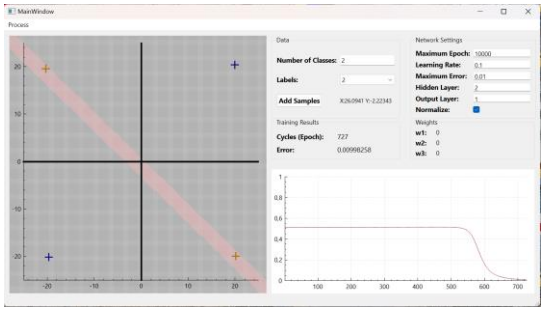
(Örnekler Grafiğe Eklendi)



(Ağırlıklar Rastgele Atandı)

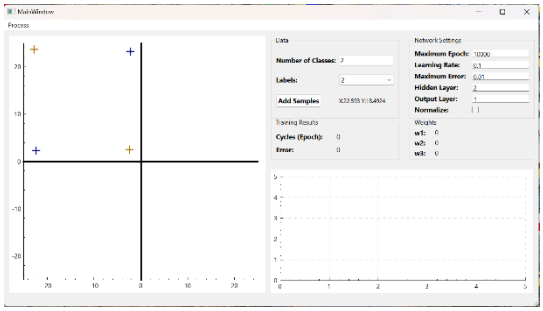


(Eğitim Normalizasyon ile Yapıldı)

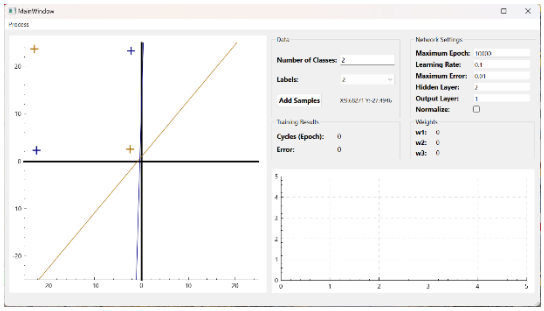


(Test İşlemi Yapıldı)

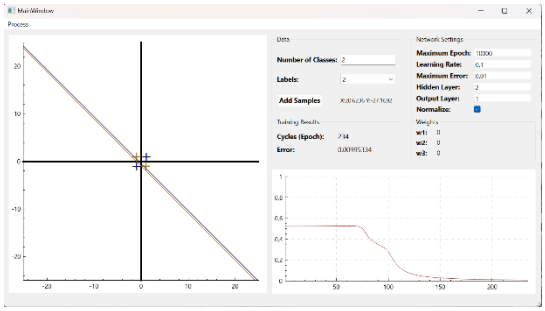
2.1 Uygulama(Delta ile Momentum Katsayısı):



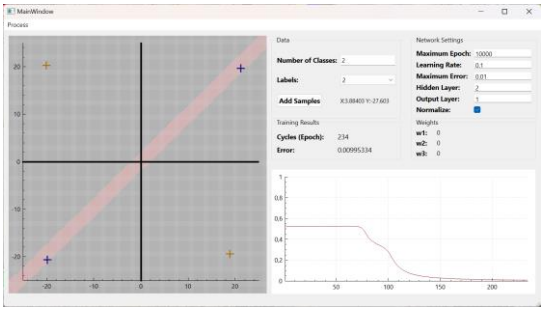
(Örnekler Grafiğe Eklendi)



(Ağırlıklar Rastgele Atandı)



(Eğitim Normalizasyon ile Yapıldı)



(Test İşlemi Yapıldı)

2.1 Uygulama Kaynak Kodları:

Delta:

```
float multi_layer_delta_rule(float* inputs, float* v, float*w, float* targets, float* hidden_bias, float* output_bias,
                             int input_dim, int hidden_size, int output_size, int num_samples, float lr)
{
    float total_error = 0.0;
    for(int i = 0; i < num_samples; i++) {
        std::vector<float> hidden_layer_output(hidden_size);

        for(int j = 0; j < hidden_size; j++) {
            hidden_layer_output[j] = 0.0;
            for(int k = 0; k < input_dim; k++) {
                hidden_layer_output[j] += inputs[i * input_dim + k] * v[k * hidden_size + j];
            }
            hidden_layer_output[j] += hidden_bias[j];
            hidden_layer_output[j] = sigmoid(hidden_layer_output[j]);
        }

        float output = 0.0;
        for(int j = 0; j < hidden_size; j++) {
            output += w[j * output_size] * hidden_layer_output[j];
        }
        output += output_bias[0];
        output = sigmoid(output);

        float d;
        if(targets[i] == 1) {
            d = -1.0;
        }
        else {
            d = 1.0;
        }
        float fnet = sigmoid_derivative(output);
        total_error += 0.5 * (d - output) * (d - output);

        for(int j = 0; j < hidden_size; j++) {
            w[j * output_size] += lr * (d - output) * fnet * hidden_layer_output[j];
        }
        output_bias[0] += lr * (d - output) * fnet;

        for(int j = 0; j < hidden_size; j++) {
            for(int k = 0; k < input_dim; k++) {
                v[k * hidden_size + j] += lr * (d - output) * fnet * w[j * output_size] * (1.0 - hidden_layer_output[j]* hidden_layer_output[j]) * inputs[i * input_dim + k];
            }
            hidden_bias[j] += lr * (d - output) * fnet * w[j * output_size] * (1.0 - hidden_layer_output[j]* hidden_layer_output[j]);
        }
    }
    total_error /= num_samples;
    return total_error;
}
```

Delta ile Momentum Katsayısı:

```
float multi_layer_delta_with_momentum(float* inputs, float* v, float*w, float* targets, float* hidden_bias, float* output_bias,
                                      double* prev_hidden_delta, double* prev_out_delta, double* prev_hidden_bias_delta, double* prev_out_bias_delta,
                                      int input_dim, int hidden_size, int output_size, int num_samples, float lr, float momentum)
{
    double total_error = 0.0;

    for(int i = 0; i < num_samples; i++) {
        std::vector<double> hidden_layer_output(hidden_size);

        for(int j = 0; j < hidden_size; j++) {
            hidden_layer_output[j] = 0.0;
            for(int k = 0; k < input_dim; k++) {
                hidden_layer_output[j] += inputs[i * input_dim + k] * v[k * hidden_size + j];
            }
            hidden_layer_output[j] += hidden_bias[j];
            hidden_layer_output[j] = sigmoid(hidden_layer_output[j]);
        }

        double output = 0.0;
        for(int j = 0; j < hidden_size; j++) {
            output += w[j * output_size] * hidden_layer_output[j];
        }
        output += output_bias[0];
        output = sigmoid(output);

        float d;
        if(targets[i] == 1.0) {
            d = -1.0;
        }
        else {
            d = 1.0;
        }
        double fnet = sigmoid_derivative(output);
        total_error += 0.5 * (d - output) * (d - output);

        for(int j = 0; j < hidden_size; j++) {
            double delta_w = lr * (d - output) * fnet * hidden_layer_output[j] + momentum * prev_out_delta[j * hidden_size];
            w[j * output_size] += delta_w;
            prev_out_delta[j * output_size] = delta_w;
        }
        output_bias[0] += lr * (d - output) * fnet + momentum * prev_out_bias_delta[0];
        prev_out_bias_delta[0] = lr * (d - output) * fnet;

        for(int j = 0; j < hidden_size; j++) {
            for(int k = 0; k < input_dim; k++) {
                double delta_v = lr * (d - output) * fnet * w[j * output_size] * (1.0 - hidden_layer_output[j] * hidden_layer_output[j]) * inputs[i * input_dim + k]
                                 + momentum * prev_hidden_delta[k * hidden_size + j];
                v[k * hidden_size + j] += delta_v;
                prev_hidden_delta[k * hidden_size + j] = delta_v;
            }
            double delta_bias = lr * (d - output) * fnet * w[j * output_size] * (1.0 - hidden_layer_output[j]* hidden_layer_output[j]) + momentum * prev_hidden_bias_delta[j];
            hidden_bias[j] += delta_bias;
            prev_hidden_bias_delta[j] = delta_bias;
        }
    }
    total_error /= num_samples;
    return total_error;
}
```