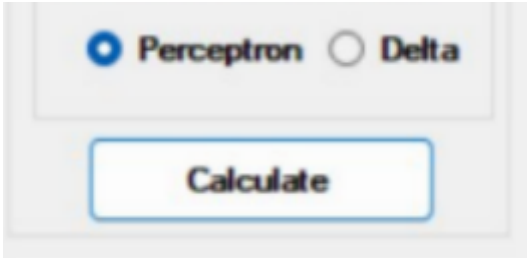
Yapay Sinir Ağları

Proje Raporu

383217

Zehra Sueda Çiğdem

Özet



Bu ödevde 2 adet fonksiyonu gerçekleştirebildim:

1) Tek nöron - 2 class
2) Tek nöron - Çok class

Her fonksiyon için Perceptron veya Delta metodlarıyla hesaplama yapmak mümkün.

Öncelikle, Perceptron ve ya Delta metodunun seçilmesinin bir yolunu yazmak istedim. Her fonksiyon için bir buton koymak yerine hangi fonksiyonu kullandığımızı belirten bir radio button çiftinin ve genel bir hesapla butonunun UX açısından daha güzel olduğuna karar verdim:

Orjinal kodda ağın (ağırlıklar, sample dizileri ve s.) ayarlanması için ayrı bir buton vardı, ben bu özelliği dropdown boxun `TextChanged()` fonksiyonuna koydum. Bu sayede class sayımızı seçtiğimiz anda ağımız hazır olacak.

Perceptron metodu

Bir perceptron yapay sinir ağlarında en basit bir nöron modelidir. Yazdığım kodda eğer Perceptron metodu kullanılacaksa, yandaki fonksiyon çağırılacaktır:

```
private:void Perceptron(float* targets) {
	bool allIsWell = false;
	while (!allIsWell) {
		allIsWell = true;

		for (int i = 0; i < numSample; i++) {
			int output = unitStepFunc(netHesaplaSingle(i, false));
			int delta = targets[i] - output;
			if (delta != 0) {
				allIsWell = false;
				agirlikGuncelle(delta, i, false);

			}
		}
	}
	LineCiz(Weights, bias, 2, 1);
}
```

Fonksiyonun çalışma mantığı, tüm sample-lar istediğimiz değeri üretene kadar ağırlıkları güncellemeye devam etmektir. Bu fonksiyon, hem 2 örnek hem de çoklu örnek için kullanılmaktadır.

$$H(x) := \begin{cases} 1, & x > 0 \\ 0, & x \leq 0 \end{cases}$$

`netHesaplaSingle()` fonksiyonu, $f(x) = w.x + b$ denklemini icra etmektedir. Kullandığımız metod Perceptron olduğu için, bu nöronun net değerini bir Unit functiona veriyoruz, bu fonksiyonun da çıktısı soldaki gibidir, yani bize ya 1 ya da 0 verir.

Ağırlıkların güncellenmesi

Bir yapay sinir ağının eğitilmesi demek, istediğimiz çıktıyı üretecek şekilde ağırlıkların ve bias-ların hesaplanmasıdır. Biz bu değerleri, incremental olarak bir döngü içerisinde hesaplıyoruz:

```
for (int i = 0; i < weightCount; i++) {

	if (normalization == true) {
		Weights[i] += delta * normalizedSamples[index * inputDim + i];
	}
	else {
		Weights[i] += delta * Samples[index * inputDim + i];
	}

}
```

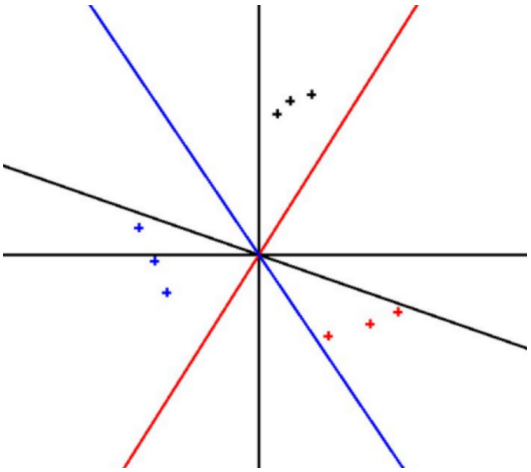
Delta metodu

Delta metodu, Perceptron-a göre daha karmaşık bir metoddur. Perceptron ile ana farkları, aktivasyon fonksiyonunun daha karmaşık olması ve normal değerler yerine normalize edilmiş değerler kullanılıyor olmasıdır:

```
float epsilon = 0.1;
error = 2;
normalizeEt();
while (error > epsilon) {
	error = 0;
	for (int i = 0; i < numSample; i++) {
		float output = sigmoidDelta(netHesaplaSingle(i, true));
		float delta = (targets[i] - output) * turevSigmoid(netHesaplaSingle(i, true));
		error += (float)0.5 * (targets[i] - output) * (targets[i] - output);
		agirlikGuncelle(delta, i, true);
	}
}
```

Bu fonksiyonda döngü hata bulunana kadar değil, her iterasyonda hesaplanan error değeri önceden belirlenmiş bir epsilon değerinden küçük olana kadar çalışacaktır. Epsilon değeri ne kadar düşük verilirse, kod o kadar uzun çalışacak ama o kadar da doğru bir değer alacaktır. Burda çözümün bulunması, bu ağın hata grafiğinde lokal "vadi"nin en alt noktasının bulunması[0] demektir. Bunun için, o grafiğin herhangi bir noktadaki türevini bulursak, grafiğin ne tarafa doğru azaldığını bulmuş oluruz ve ağdaki değerleri ona göre güncelleyebiliriz.

İkiden fazla class



Eğer bizim kodumuzun ikiden fazla classları ayırması gerekiyorsa, bunu yapmak için ya her bir class için bir nöron olacak şekilde multi-layer multi-neuron bir yapı yapabiliriz, ya da tek nöronu her classı diğerlerinden ayıracak şekilde eğitip ağırlıkları tutabiliriz. Bunun için, ana fonksiyon başladığında eğer ikiden fazla class varsa, bir döngü içerisinde `fakeTargets` dizisi içinde her class için o classın değerlerini 1, diğerlerini 0 yapıyoruz, ve normalde iki classı ayırmak için tasarlanan Perceptron ve Delta fonksiyonlarımıza bu `fakeTargets` dizisini gönderiyoruz. Her class için bunu ayrıca yaparsak, sonuçta tüm classlarımız için onu diğer classlardan ayıran ağırlık ve bias değerlerini buluyor ve bunu `finalWeights[]` dizisinde tutuyoruz. Orjinal kodda `Weights[]` dizisi daha jenerik bir dizi olarak tasarlanmıştı, ancak ben bu kodumda bu diziyi sadece iki classı ayırmak için `Weights[0]` ve `Weights[1]` değerlerini tutacak şekilde kullanıyorum.

Sonuçların ekrana çizdirilmesi

Orijinal kodda `LineCiz(...)` fonksiyonu, tüm ağırlıklar hesaplandıktan sonra tek bir şekilde çağrılacak ve gerekirse tek gerekirse çoklu çizgileri yazdıracak şekilde tasarlanmış bir fonksiyondu. Ancak ben mutlilayer-multineuron yapmadığım ve kodumun temelde sadece iki classı ayırmak fonksiyonları üzerinden çalışmazı yüzünden, bu metodu her bir çizgi çizileceği zaman çağırılan bir fonksiyon olarak düzenledim. Hangi renkte yazdıracağını da, global olarak tutulan `currentClass` değişkeninden anlıyor.Bu çıktıdaki döngü sayısı, her class için sıfırlanmıyor. Yani soldaki ornekte 1. Class için 3, 2. Class için 1, 3. Class için 2 döngü yapılmıştır:

Son olarak, ekrana hesaplanan değerleri yazdırdım. Hesaplanırken anlık değerlerin gösterilmesi de mümkün, ama bu performansı çok etkilediği için bir checkbox arkasına koydum.

Bu ödevde git de kullandım. Kodlarımı ve commit geçmişimi Github hesabımdan[1] görebilirsiniz.

[0]: https://standoutpublishing.com/g/local%20minimum.html
[1]: https://github.com/suedac/neural_network/