Relatorio sobre Backpropagation em redes neurais multicamadas

Aluno: Luiz Cortinhas Ferreira Neto

1. **Introdução**

Inspirados pela funcionalidade sofisticada do cérebro humano com seus milhões de neurónios interconectados e processando informação de forma paralela nasce o conceito computacional de rede neural multi-camadas baseada no perceptron (Wang,2003): Rede artificial organizada em camadas, suas camadas contem partes menores (neurónios) são capazes de transferir e/ou armazenam o conhecimento do padrão percebido em sua entrada.

Por sua arquitetura, as redes neurais artificiais multi camadas baseadas em perceptron - MLP possuíram durante muito tempo um gap cientifico que resplandecia sob o processo de propagação de erros, portanto treinamento, da rede. Quando em 1970, surgem os primeiros artigos que apresentam a solução de retro-propagação de erros por calculo diferencial (S. Linnainmaa,1970), comummente chamado de back-propagation. Hoje, depois de 60 anos de sua primeira publicação, o algoritmo possui aperfeiçoamentos e alternativas como o Modified Hebbian Learning Rule (Bechtel, W., & Abrahamsen, A.,1993).

1. **Domínio e Dados utilizados**

Por familiaridade e habilidade prévia, o domínio escolhido para este experimento é o sensoriamento remoto optico. O sensoriamento remoto é a utilização de sensores para aquisição de informações sobre objetos ou fenómenos sem que haja contato direto entre eles. Os dados provenientes que serão utilizados compõem um dataset do projeto Mapbiomas (www.mapbiomas.org) para classificação de corpos de agua na superfície terrestre.

O dataset aqui utilizado é dividido em duas partes: treino com 30 amostras e teste com 20 amostras, cada instância do dataset possui três atributos, conhecidos no domínio como indices espectrais, representam a operação algébrica entre as bandas de um satélite optico para melhor deteção espectral de: Vegetação (NDVI) e Corpos de Agua (NDWI), alem uma banda optica no infra-vermelho próximo (SWIR1).

Para facilitar a convergencia futura da rede, é importante citar que o dataset inteiro foi normalizado para o intervalo de 0 a 1.

1. **Implementação - JavaScript**

// Neuron Object Prototype

function neuron(name) {

this.name = name;

this.numberOfConnections= 2;

this.input= [];

this.weights= [];

this.bias= 1;

this.output= 0;

this.activationFunction= function(){

if(!Array.isArray(this.input)){

this.input = [this.input]

}

this.numberOfConnections = this.input.length;

var sum = 0;

for(var i = 0; i < this.numberOfConnections; i++){

sum = sum+((this.input[i] \* this.weights[i]) + this.bias); // Activation Function

}

this.output = sum;

return this;

};

this.weightsInit = function(){

if(!Array.isArray(this.input)){

this.input = [this.input]

}

if(this.input.length === 0){

console.log('ERROR: no Inputs, you must insert the input first');

}

this.numberOfConnections = this.input.length;

for(var i = 0; i < this.numberOfConnections; i++){

this.weights[i] = Math.random();

}

return this;

};

return this;

};

// Fully Connected Layer Object Prototype

function FullyConnectedLayer() {

this.name= "default";

this.neurons=[];

this.inputLayer=[]; // after Layer OUT

this.outputs=[];

this.createLayer = function(numberOfNeurons,inputLayer,name,first){

this.name = name;

this.neurons = [];

this.inputLayer = inputLayer;

for(var i = 0; i < numberOfNeurons;i++){

this.neurons[i] = new neuron(name+'-'+i);

if(first){

this.neurons[i].input = inputLayer[0][i];//this.inputLayer.outputs();

this.neurons[i].numberOfConnections = inputLayer[0][i].length;

this.neurons[i].weightsInit();

}else{

this.neurons[i].input = inputLayer;//this.inputLayer.outputs();

this.neurons[i].numberOfConnections = numberOfNeurons;

this.neurons[i].weightsInit();

}

}

return this;

};

//This function execute every neuron saving each output

this.runLayer = function(){

if(this.neurons.length === 0){

console.log("First, you must create the Layer");

return 0;

}

this.output = [];

for(var i = 0; i < this.neurons.length;i++){

this.neurons[i] = this.neurons[i].activationFunction();

this.output[i] = this.neurons[i].output;

console.log('Neuron',this.neurons[i].name, this.output[i]);

}

return this;

};

return this;

};

// Multi Layer Perceptron Object Prototype

function Network() {

this.layers=[];

this.input=[];

this.desirable=[];

this.output=[];

this.learningRate= 0.1;

this.init= function(nLayers,numberOfNeurons,dataset){

if(dataset.values.length === 0 || dataset.desirable.length === 0){

console.log('Check you train dataset, something is wrong!');

return 0;

}

if(dataset.values.length != dataset.desirable.length){

console.log('Check you train dataset, desirable and values must have the same length!');

return 0;

}

this.input = dataset.values;

this.desirable = dataset.desirable;

//Fully Connected Layers Creation :)

for(var born = 0; born < nLayers;born++){

if(born == 0){ // FIRST LAYER

this.layers[born] = new FullyConnectedLayer().createLayer(numberOfNeurons,dataset.values,"startLayer",true);

}else{

//Note: for the instance, every input is [empty]

this.layers[born] = new FullyConnectedLayer().createLayer(numberOfNeurons,Array.apply(null, Array(numberOfNeurons)).map(Number.prototype.valueOf,0),"layer-"+born,false);

}

}

console.log("Layers Created");

return this;

}

//Feed(train) or RUN(predict)!

this.feed = function(){

for(var i = 0; i < this.layers.length;i++){

console.log('Camada:',this.layers[i].name);

this.layers[i] = this.layers[i].runLayer();

if((i+1) < this.layers.length){

this.layers[i+1].inputLayer = this.layers[i].output;

}

}

return this;

}

}

Neuron.prototype.calculate\_output\_gradients = function(target){

// Calculate the difference between our target and the output we got

var delta = target - this.output\_value;

// Find the difference between them

this.gradient = delta \* this.transfer\_function\_derivative(this.output\_value);

//console.log("Updated output gradient to: " + this.gradient);

};

Neuron.prototype.calculate\_hidden\_gradients = function(nextLayer){

// Sum the next layer

var sum = 0;

for(var n = 0; n < nextLayer.length - 1; n++){

//console.log(nextLayer[n].get\_gradient());

sum += this.output\_weights[n].get\_weight() \* nextLayer[n].get\_gradient();

}

// Calculate gradients

this.gradient = sum \* this.transfer\_function\_derivative(this.output\_value);

};

NNetwork.prototype.back\_prop = function(target){

// Start with grabbing our output layer

var output = this.layers[this.layers.length - 1];

// Calcualte the differences between what the NN got and what we wanted the

// NN to get.

for(var n = 0; n < output.length - 1; n++){

output[n].c(target[n]);

}

// Go backwards through the network and update values we expected vs values

// we got. This allows our NN to "learn".

for(var l = this.layers.length - 2; l > 0; l--){

var hiddenLayer = this.layers[l];

var nextLayer = this.layers[l+1];

for(var n = 0; n < hiddenLayer.length; n++){

hiddenLayer[n].calculate\_hidden\_gradients(nextLayer);

}

}

// Go backwards again and update the inputs to make sure we can reflect

// these changes.

for(var l = this.layers.length - 1; l > 0; l--){

var currentLayer = this.layers[l];

var prevLayer = this.layers[l - 1];

for(var n = 0; n < currentLayer.length - 1; n++){

currentLayer[n].update\_input\_weights(prevLayer);

}

}

}

//Instance Test

var nn = new Network();

nn = nn.init(2,4,DataSetSingle);

console.log('Antes',nn.layers);

nn = nn.feed();

console.log(nn.layers);

1. **Resultados Experienciados**

A execuçao do script acima prevê uma configuração de rede baseada em 2 neurónios de entrada, uma camada intermediaria totalmente interconectada e uma camada de output baseada em Sigmoide, conforme apresentado na Figura 1.

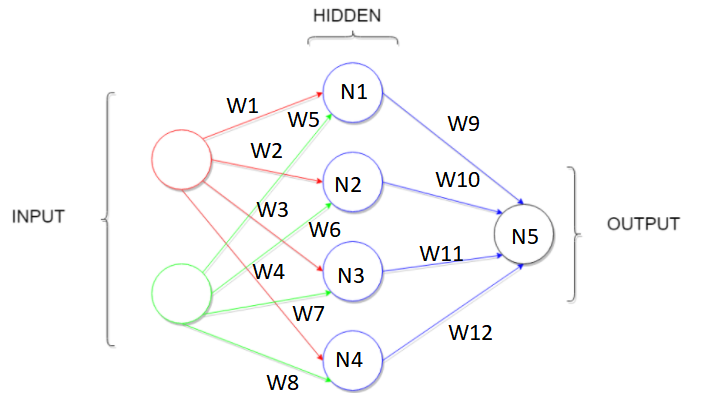
****

Figura 1 - Arquitetura de Rede Gerada

Os pesos foram inicializados de maneira aleatória, de 0 a 1, após isso é executada a função de feed foward para cada instância do dataset, a operação de feed-foward é traduzida pela formulação apresentada na Figura 2, esta formulação é convertida ao loop de somatório de todas as formulas bases do neurónio ( peso \* entrada neurónio + bias) contido na função ***runLayer(),*** sendo propagado pelo loop contido na função ***feed()****.*

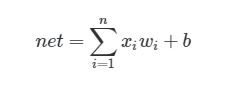


Figura 2 - Representação matemática do feed-forward

Por padrão desta implementação, todos os neurónios da rede, estão programados de forma estática para reagir a função de ativação Sigmoid, portanto pensados para dar resposta entre 0 e 1 tendendo aos limites, suficientes para responder a um problemas de classificação binária.

Ao final da etapa de feed, que ocorre para cada instância do treinamento, a rede deve calcular o erro entre o predito e o esperado (supervisionado), a diferença entre estes valores chamamos de erro também deve-se calcular o erro delta responsável por apresentar a variação possível de erro encontrada ao final de cada interação, conforme formula exposta na figura 3.

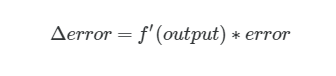


Figura 3 - Formula para calculo de erro delta

Neste ponto do algoritmo, agora inicia-se a função ***back\_prop()*** que de maneira interativa, camada a camada, é responsável por dar inicio a tarefa de backpropagation e distribuir ajustes aos pesos das sinapses de maneira proporcional ao erro delta associado, como demonstrado na Figura 4 existem para nossa arquitetura duas variações da formula: a primeira corresponderá a primeira camada (da direita para esquerda, portanto pesos de W9 a W12 na Figura 1 acima) considerando o erro delta associado a resposta de ***sigmoid(N5)*** e a segunda formula para os pesos de (W1 a W8) considerando o erro associado a resposta da camada escondida, calculados por decomposição por derivação.

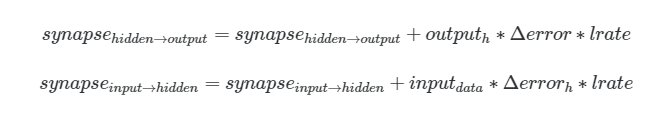


Figura 4 - Equações para ajuste dos pesos

Importante notar: Para esta rede neural o learning rate é baseado em gradiente, portanto seu valor inicial é de 0.1 podendo chegar a 0.000001.

Ao final do processo de back-propagation e ajuste dos pesos, a rede inicia novamente o feed-foward para uma nova instância do dataset e reinicia o ciclo de auto ajuste, ate que a condição de parada seja satisfeita, neste caso, parada manualmente.

Na figura 5, temos o erro médio quadrático encontrado para cada época, considerando uma época a volta completa no dataset de treinamento. Seguindo para Tabela 1 que mostra os resultados consolidados no formato de tabela de contingência, contendo nas colunas o predito e nas linhas o esperado.

Figura 5 - Erro médio quadrático por interação da rede

|  |  |  |
| --- | --- | --- |
|  | **0** | **1** |
| **0** | 17 | 3 |
| **1** | 2 | 8 |

Tabela 1 - Resultado da predição, após 23 épocas de treinamento

1. **Conclusão**

A acuracia do treinamento após as 23 épocas é de 83%, o algoritmo backpropagation demonstra exatidão ao possibilitar a majoritaria inferencia do dataset, demonstrando capacidade de operar com dados não filtrados vindos de valores de pixeis extraídos de imagens de satélite. Contudo é diante de sua implementação é notável que quanto mais horizontal é uma rede neural mais recursos de maquina devem ser consumidos, alem do fato, de que o algoritmo é bastante recursivo, podendo talvez inviabilizar a sua utilização em menor poder de processamento como os embarcados, obviamente isto não diminui suas qualidades apenas é um indicativo para sua escolha em aplicações com maior disponibilidade de recursos computacionais.

1. **Referencia**
2. Linnainmaa. The representation of the cumulative rounding error of an algorithm as a Taylor expansion of the local rounding errors. Master's Thesis (in Finnish), Univ. Helsinki, 1970. See chapters 6-7 and FORTRAN code on pages 58-60. [PDF](http://www.idsia.ch/~juergen/linnainmaa1970thesis.pdf). See also BIT 16, 146-160, 1976.
3. Bechtel, W., & Abrahamsen, A. (1993). Connectionism and the mind. Oxford, UK: Blackwell.