Neural Networks

jkmiller

November 2022

1 Introduction

Let L be the number of layer, and $n_1, \dots n_L$ be the number of neurons in each layer of our network. Let n_0 denote the number of dimensions of the input. For each layer $\ell \in \{1, \dots, L\}$ we have a matrix of weights $W^\ell \in M(n_\ell, n_{\ell-1})$ and a vector of biases $B^\ell \in M(n_\ell, 1)$. The activation of a neuron $i \in \{1, \dots, n_\ell\}$ in layer $\ell \in \{1, \dots, L\}$ is defined to be the number a_i^ℓ given by

$$a_i^\ell = \sigma\left(\sum_{j=1}^{n_{\ell-1}} w_{i,j}^\ell a_j^{\ell-1} + b_i^\ell\right) = \sigma(z_i^\ell) \tag{1}$$

or in vector notation

$$A^\ell = \boldsymbol{\sigma}\left(W^\ell A^{\ell-1} + B^\ell\right) = \boldsymbol{\sigma}(Z^\ell) \in M(n_\ell, 1). \tag{2}$$

We note that in particular,

$$Z^\ell \in M(n_\ell, 1). \tag{3}$$

Here, we are letting $\sigma : \mathbb{R} \to \mathbb{R}$ be the non-linearity, and $\boldsymbol{\sigma} : \mathbb{R}^d \to \mathbb{R}^d$ be the "vectorized" non-linearity $(x_1, \dots, x_d) \mapsto (\sigma(x_1), \dots, \sigma(x_d))$. The input is defined to be the "zero-th" level activation:

$$A^0 = input, \tag{4}$$

and the output of the network is defined to be

$$N(A^0) = A^L. \tag{5}$$

The derivative of the loss function will now be computed inductively using so called "back-propagation". Let us define for each $A^* \in \mathbb{R}^{n_L}$ the L^2 based loss function

$$\mathcal{L}(A^*) = \frac{1}{2}\sum_{j=1}^{n_L}(a_j^L - a_j^*)^2 \tag{6}$$

Then we can compute, using the chain rule and Einstein notation, that for each $\ell \in \{1, \dots, L\}$ we have the gradient of the loss with respect to the biases are

$$\frac{\partial \mathcal{L}}{\partial b_i^\ell} = \frac{\partial \mathcal{L}}{\partial a_k^\ell} \frac{\partial a_k^\ell}{\partial b_i^\ell} \tag{7}$$

$$= \frac{\partial \mathcal{L}}{\partial a_k^\ell} \frac{\partial}{\partial b_i^\ell} \sigma \left(\sum_{m=1}^{n_{\ell-1}} w_{k,m}^\ell a_m^{\ell-1} + b_k^\ell \right) \tag{8}$$

$$= \frac{\partial \mathcal{L}}{\partial a_k^\ell} (\sigma'(z_k^\ell) \delta_{i,k}) \tag{9}$$

$$= \sigma'(z_i^\ell) \frac{\partial \mathcal{L}}{\partial a_i^\ell}. \tag{10}$$

This quantity ends up being used throughout the computations below, so it is useful to compute it first. Next, we compute the gradient with respect to the weights

$$\frac{\partial \mathcal{L}}{\partial w_{i,j}^\ell} = \frac{\partial \mathcal{L}}{\partial a_k^\ell} \frac{\partial a_k^\ell}{\partial w_{i,j}^\ell} \tag{11}$$

$$= \frac{\partial \mathcal{L}}{\partial a_k^\ell} \frac{\partial}{\partial w_{i,j}^\ell} \sigma \left(\sum_{m=1}^{n_{\ell-1}} w_{k,m}^\ell a_m^{\ell-1} + b_k^\ell \right) \tag{12}$$

$$= \frac{\partial \mathcal{L}}{\partial a_k^\ell} \left(\sigma'(z_k^\ell) \delta_{i,k} a_j^{\ell-1} \right) \tag{13}$$

$$= a_j^{\ell-1} \sigma'(z_i^\ell) \frac{\partial \mathcal{L}}{\partial a_i^\ell} \tag{14}$$

$$= a_j^{\ell-1} \frac{\partial \mathcal{L}}{\partial b_i^\ell}. \tag{15}$$

Finally, we can compute the gradient with respect to the activation of the a lower level

$$\frac{\partial \mathcal{L}}{\partial a_j^{\ell-1}} = \frac{\partial \mathcal{L}}{\partial a_k^\ell} \frac{\partial a_k^\ell}{\partial a_j^{\ell-1}} \tag{16}$$

$$= \frac{\partial \mathcal{L}}{\partial a_k^\ell} \frac{\partial}{\partial a_j^{\ell-1}} \sigma \left(\sum_{m=1}^{n_{\ell-1}} w_{k,m}^\ell a_m^{\ell-1} + b_k^\ell \right) \tag{17}$$

$$= \sum_{k=1}^{n_\ell} \frac{\partial \mathcal{L}}{\partial a_k^\ell} \sigma'(z_k^\ell) w_{k,j}^\ell, \tag{18}$$

$$= \sum_{k=1}^{n_\ell} \frac{\partial \mathcal{L}}{\partial b_k^\ell} w_{k,j}^\ell, \tag{19}$$

We can write the above formulae in vector form using the *Hadamard product*. Given two matrices A and B of the same shape, we define the Hadamard product

component by component as

$$(A \odot B)_{i,j} = A_{i,j} B_{i,j}. \tag{20}$$

Note that the above product is commutative, unlike the standard matrix product. Recall also the *tensor product* of two vectors v, w of the same length:

$$(v \otimes w)_{i,j} = v_i w_j. \tag{21}$$

Note that the tensor product is *not* commutative. Now, using the obvious shorthand notation for the gradients, we may write

$$\boxed{\nabla_B^\ell \mathcal{L} = A^{\ell-1} \odot \nabla_A^\ell \mathcal{L}} \tag{22}$$

$$\boxed{\nabla_W^\ell \mathcal{L} = \nabla_B^\ell \mathcal{L} \otimes A^{\ell-1}} \tag{23}$$

$$\boxed{\nabla_A^{\ell-1} \mathcal{L} = (\nabla_B^\ell \mathcal{L})^t W^\ell} \tag{24}$$

Using these formula, we may compute gradient of $\mathcal{L}$ inductively:

1. Start with $\ell = L$. Then, by the definition of the L^2 loss we have

$$\nabla_A^L \mathcal{L} = A^L - A^* \tag{25}$$

2. For $\ell = L, \ldots, 2$ compute $\nabla_B^\ell \mathcal{L}, \nabla_W^\ell \mathcal{L}$, and $\nabla_A^{\ell-1} \mathcal{L}$.

3. Finally, compute $\nabla_B^1 \mathcal{L}, \nabla_W^1 \mathcal{L}$. This step requires the input layer.

After these derivatives have been computed (and perhaps averaged over a training set), one can carry out a gradient descent algorithm or stochastic gradient descent algorithm on the weighs and biases. For example, given a step size $\gamma > 0$ which is sufficiently small, we may update the weights via

$$W^\ell \mapsto W^\ell - \gamma \nabla_W^\ell \mathcal{L} \tag{26}$$

$$B^\ell \mapsto B^\ell - \gamma \nabla_B^\ell \mathcal{L} \tag{27}$$

2 Implementation in Python

Let us implement the above framework in Python. First, we load the required libraries and define functions involving our non-linear function σ.

```
import math
import numpy as np

# These are constants which are better to precompute
pii = 1/math.pi
```

```
def sigma(x):
    """This defines the nonlinearity for each neuron. Range of function
        is (0,1)."""
    return np.arctan(x)*pii + 0.5

def d_sigma(x):
    """This is the derivative of the nonlinearity"""
    return pii/(1+ (x ** 2))
```

Next, we define the general architecture of our network. For example, we have picked $L = 3$, with $n_0 = 3, n_1 = 2, n_2 = 3$, and $n_3 = 4$. Note that $L + 1$ is the *network depth* below.

```
neuron_parameters = [3,2,3,4]
network_depth = len(neuron_parameters)

#Random network initialization as a list
network_weight = []
network_bias = []
for n in range(1,network_depth):
    network_weight.append(np.zeros([neuron_parameters[n],neuron_parameters[n-1]]))
    network_bias.append(np.zeros([neuron_parameters[n]]))

#Network activation and z initialized as zeros
network_activation = []
network_z = []
for n in range(1,network_depth):
    network_activation.append(np.zeros([neuron_parameters[n]]))
    network_z.append(np.zeros([neuron_parameters[n]]))
```

Note that lists in Python start at 0, so the network weights are labeled $0, 1, 2$ (think of the lines between the nodes of the network), the biases/activations/z's labeled $0, 1, 2$ (think of node layers not including the data layer). The last activation layer, which is the output of the network, is labeled $L - 1 =$ network depth $- 2$, which is 2 in our example.

With the network architecture initialized, we can now compute the output of the network on a sample input.

```
#Computes the output of the network given input_data
input_data = np.array([1,1,1])

network_z[0] = np.matmul(network_weight[0],input_data) + network_bias[0]
network_activation[0]= sigma(network_z[0])

for n in range(1,network_depth-1):
    network_z[n] =
        np.matmul(network_weight[n],network_activation[n-1])+network_bias[n]
    network_activation[n] = sigma(network_z[n])
```

Next, we can compute the gradients via back-propagation comparing to some *output data*.

```
#Computes the L2 gradient of the loss by comparing with output_data
output_data = np.array([2,2,2,2])

gradient_activation[network_depth-2] =
    network_activation[network_depth-2]-output_data
for n in range(network_depth-2,0,-1):
    gradient_bias[n] = (d_sigma(network_z[n]))*(gradient_activation[n])
    gradient_weight[n] =
        np.tensordot(gradient_bias[n],network_activation[n-1],axes = 0)
    gradient_activation[n-1] =
        np.matmul(np.transpose(gradient_bias[n]),network_weight[n])

gradient_bias[0] = d_sigma(network_z[0])*gradient_activation[0]
gradient_weight[0] = np.tensordot(gradient_bias[0],sample_data,axes = 0)
```

Finally, we may update the weights and biases via gradient descent, with some rate $\gamma > 0$. A common choice is $\gamma = 0.001$.

```
#Update the weights and biases via gradient descent
gamma = 0.001
for n in range(network_depth-1):
    network_weight[n] = network_weight[n] - gamma*gradient_weight[n]
    network_bias[n] = network_bias[n] - gamma*gradient_bias[n].
```