CS419M

Lecture 20: Introduction to Deep Learning

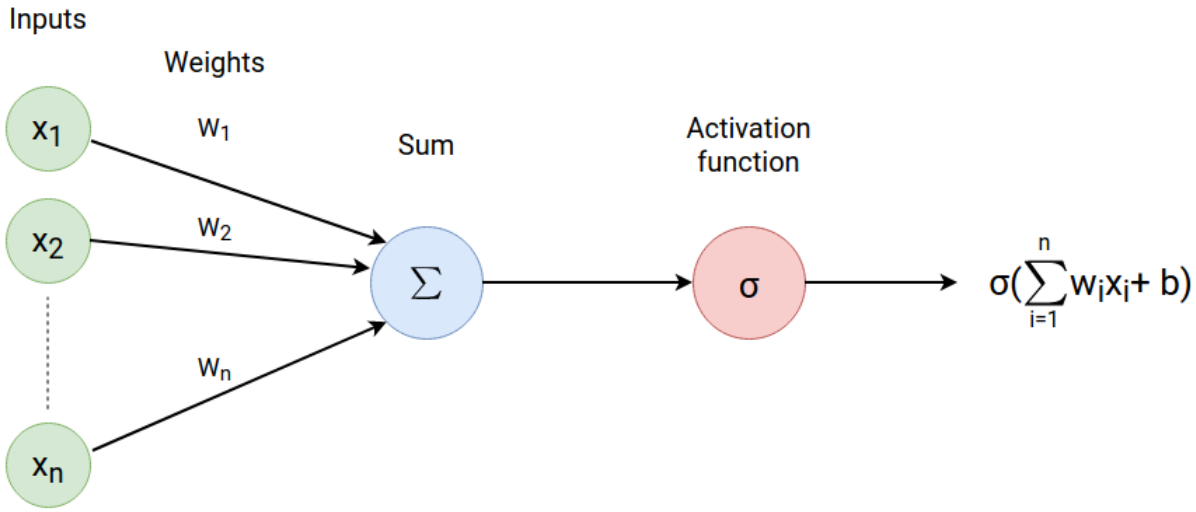
5 April 2023

Lecturer: Abir De *Scribe: Group 22*

1 Introduction

1.1 Capacity of the perceptron

What kind of function can a neural network represent ?
Let's start with the simplest neural network, the Perceptron. It is effectively a NN with a single hidden layer having 1 hidden unit with an activation function σ. Both the input layer and the weights are a $1 \times$ n vector.



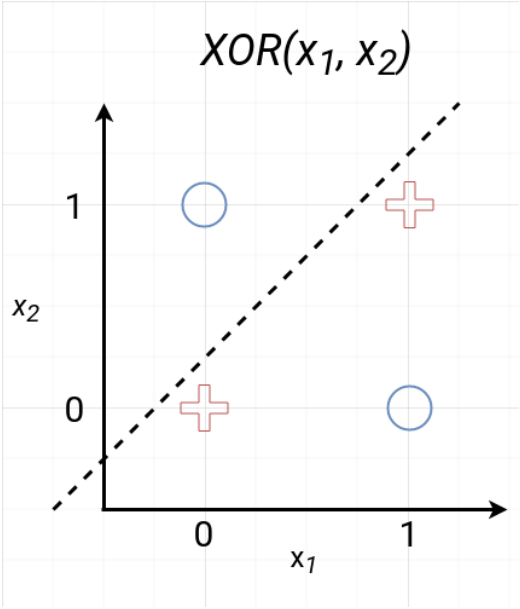
The perceptron can model a function of this form: $\sigma(\sum_i^n \mathbf{w_i} x_i + b) = \sigma(\mathbf{w}^\top \mathbf{x} + b)$. Often, we select the non-linear function σ to be one of the following:

$$\text{sigmoid} = \frac{1}{1 + e^{-x}}$$

$$\tanh = \frac{e^{2x} - 1}{e^{2x} + 1}$$

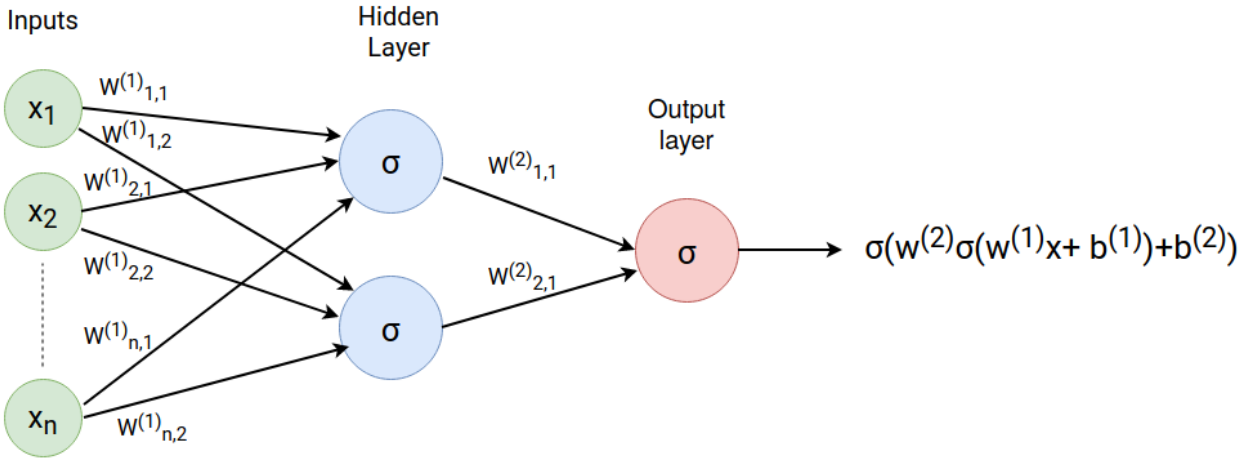
$$\text{ReLU} = \max(0, x)$$

Notably, the Perceptron is a linear classifier, and as such it famously can't model an XOR.



1.2 Capacity of multiple neurons

By allowing ourselves more than 1 neuron in the hidden layer, we can model a XOR and in fact, we get the simplest **universal approximator**.



2 Universal Approximation theory

The universal approximation theorem states that any continuous function $f : [0, 1]^n \to [0, 1]$ can be approximated arbitrarily well by a neural network with at least 1 hidden layer with a finite number of weights.

3 Deep Learning

3.1 Goal

We are trying to model:

$$y = w^T \phi(x)$$

but ϕ can be any non linearity. So how to model it to cover all types of functions?

Take a linear function: f(x) = $A^{m \times d}$ x

Now let's apply a non-linear function g such as relu, sigmoid or tanh to all the points and get a vector. We get:

$$g(Ax) = Relu(Ax) \; or \; tanh(Ax) \; or \; \sigma(Ax)$$

Next, let's apply another linear function h. This can be achieved by multiplying it by a matrix B. So we get:

$$h(g(Ax) = Bg(Ax)$$

As per the universal approximation theorem, If non linear function is fixed, then there exists an A and B which can model any arbitrary non linear function.
'm' will determine how close it is to the original function.

$$\|Bg(Ax) - w^T \phi(x)\| < \epsilon$$

3.2 Algorithm

Our goal is to minimize the following loss function:

$$\min_{A,B} \sum (y_i - Bg(Ax))^2$$

So we start with random A and B and perform gradient descent:

$$A_{t+1} \leftarrow A_t - r\Delta_A \, l(A, B)|_{A_t, B_t}$$
$$B_{t+1} \leftarrow B_t - r\Delta_B \, l(A, B)|_{A_t, B_t}$$

To deal with multiple minimas and to find the global minima, we repeat the process with another set of A and B multiple times. in the end, we compare losses from all sets and choose the minimum one.

for $t \in 1 \dots T$ **do**
  $loss \leftarrow 0$
  for $i \in D$ **do**
    $loss+ = (y_i - Bg(Ax))^2$
    $L \leftarrow loss$
    $A, B \leftarrow GradientDescent(L)$
  end for
end for

However, we run into a problem here. Since we are using for loop, it will take a tremendous amount of time to calculate the loss. Because we are calculating loss for individual points. This can be avoided if we tensorize the loss. and calculate the loss matrix and sum it as follows:

$$L = (Y - Bg(AX))^2$$
$$L = L.sum()$$

However we run into another problem as this we require a lot of memory. So to rectify this we tensorize in parts. This process is called batching and it allows for best utilization of memory

3.3 Batching

3.3.1 How to modify the algorithm

Divide the dataset into random batches. For doing so, we can use numpy function numpy.random .shuffle(). Dimension of X is taken as N x d, batch size = b. So the modified code would look like

for $t \in 1$ to T **do**
  $loss \leftarrow 0$
  $Index \leftarrow$ Permute(1..dim(Y))
  $X' \leftarrow$ X[Index]
  for $i \in 0$ to $\lfloor N/b \rfloor - 1$ **do**
    $X" \leftarrow X'$[ib:(i+1)b]
    $Y" \leftarrow Y'$[ib:(i+1)b]
    $L \leftarrow (Y" - B(g(AX")))^2$.sum()
    $A, B \leftarrow$ gradDes(L)
  end for
end for

But why we are randomly splitting the dataset to take one batch from it?
If we perform gradient descent on entire dataset it will be slow. Instead we are permuting the dataset, taking a batch from it and performing gradient descent on that batch. It is faster. To do so,

we need to keep in mind that the batch that we are selecting should represent the entire dataset. If we choose a batch without permuting the data, it might overfit one particular batch. On the other hand, if we are performing gradient descent on the entire dataset it might underfit.

4 References

[1] http://mitliagkas.github.io/ift6085-2020/ift-6085-lecture-10-notes.pdf