CS 217: Artificial Intelligence and Machine Learning **Jan-Apr 2024**

Lecture 11: Introduction to Neural Networks

Lecturer: Swaprava Nath *Scribe(s): SG22*

Disclaimer: *These notes aggregate content from several texts and have not been subjected to the usual scrutiny deserved by formal publications. If you find errors, please bring to the notice of the Instructor.*

11.1 Philosophy behind Neural Networks

Consider the example of points shown below in the diagram; there are some cross($\times$) symbolic points and some circular ($\circ$) points, and we have to classify them into two different classes.

If we use *logistic regression* for classification, as the decision boundary of logistic regression is linear (because

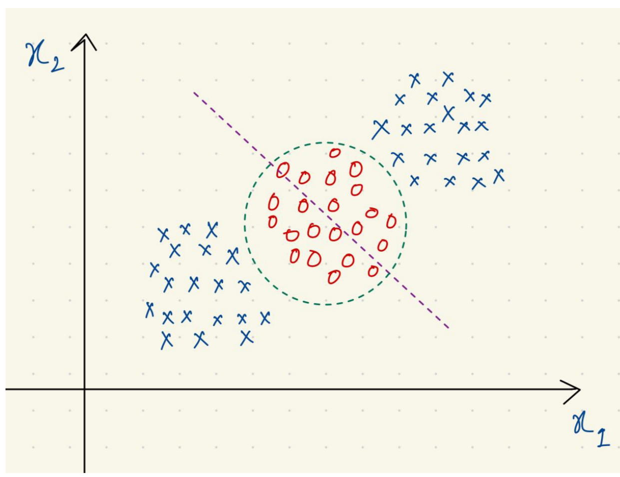


Figure 11.1: Two class of points

we are using $w^T X$ as our boundary, which is linear in 2-D) in the 2-D case, it is not possible to perfectly classify the points into two different classes as a line can't separate the $\times$ and $\circ$ points into two different regions.

$$f(X, w) = \frac{1}{1 + e^{-w^T X}} \text{ (Logistic Regression)}$$

So, to solve this problem of the linear decision boundary, we can use the concept of the basis function, which gives us a non-linear decision boundary.
Let us take the basis function as :

$$\Phi(X) = \begin{pmatrix} 1 & x_1 & x_2 & x_1^2 & x_2^2 \end{pmatrix}^T$$

Now, the decision boundary of this basis function is circular, and by properly adjusting the weights, we could find a circular decision boundary that contains all the $\circ$ points.

$$f(\Phi(X), w) = \frac{1}{1 + e^{-w^T \Phi(X)}} \text{ (Logistic Regression with basis function)}$$

In Neural Networks, our goal is to attain non-linear behaviour without the need for explicit programming dedicated to non-linearity; this is the fundamental principle behind Neural Networks.

11.2 Back in Time: A Quick Look at History

- 1943 - Neurophysiologist Warren McCulloch and mathematician Walter Pitts wrote a paper on how neurons might work. In order to describe how neurons in the brain might work, they modelled a simple neural network using electrical circuits. They also introduced the perceptron concept.
- 1950s- Nathanial Rochester from the IBM research laboratories led the first effort to simulate a neural network. Unfortunately for him, the first attempt to do so failed.
- 1957- The first hardware implementation of perceptron was Mark I Perceptron machine built in 1957 at the Cornell Aeronautical Laboratory by psychologist Frank Rosenblatt, funded by the Information Systems Branch of the United States Office of Naval Research and the Rome Air Development Center.
- 1982- Interest in the field was renewed. John Hopfield of Caltech presented a paper to the National Academy of Sciences. His approach was to create more useful machines by using bidirectional lines. Previously, the connections between neurons were only one way.
- 1982- US-Japan Joint Conference on Cooperative/ Competitive Neural Networks at which Japan announced their Fifth-Generation effort resulted US worrying about being left behind. Soon, funding was flowing once again.
- 1997- A recurrent neural network (RNN) framework, Long Short-Term Memory (LSTM), was proposed by Schmidhuber & Hochreiter.
- 1998- Yann LeCun published Gradient-Based Learning Applied to Document Recognition.

Now, discussions on neural networks are prevalent; the future is here!

11.3 Artificial Neural Networks

First, I want us to understand why neural networks are called neural networks. The way an actual neuron

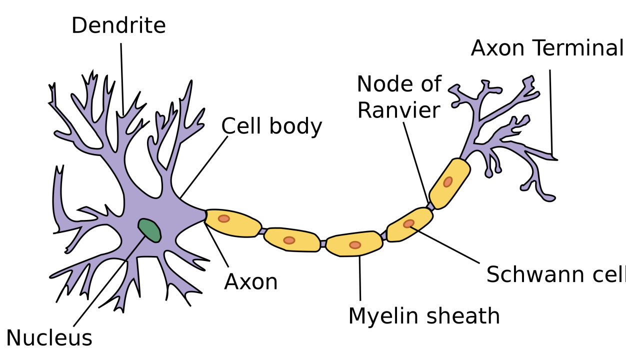


Figure 11.2: Diagram of a Neuron

works involves the accumulation of electric potential, which, when exceeding a particular value, causes the pre-synaptic neuron to discharge across the axon and stimulate the post-synaptic neuron. The human brain's capabilities are incredible compared to what we can do even with state-of-the-art neural networks.
We can draw a neural diagram that makes the analogy between the neuron structure and the artificial neurons in a neural network.

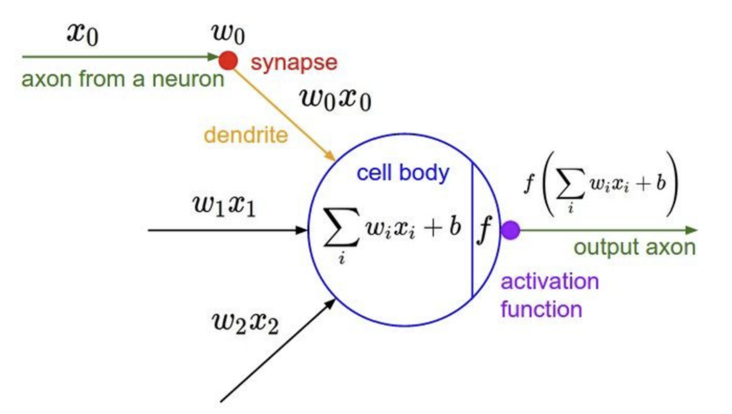


Figure 11.3: Diagram of an Artificial Neuron

If we have multiple features, each is passed through an affine transformation, which is basically a weighted sum of input features with some bias term, which gives us something resembling a regression equation.
We then pass this result through our activation function, which gives us some form of probability. This probability determines whether the neuron will fire — our result can then be plugged into our loss function to assess the algorithm's performance.
Here is a multi-layer neural network, and our target is to learn the Ws and the bs of all layers to minimize the loss J.

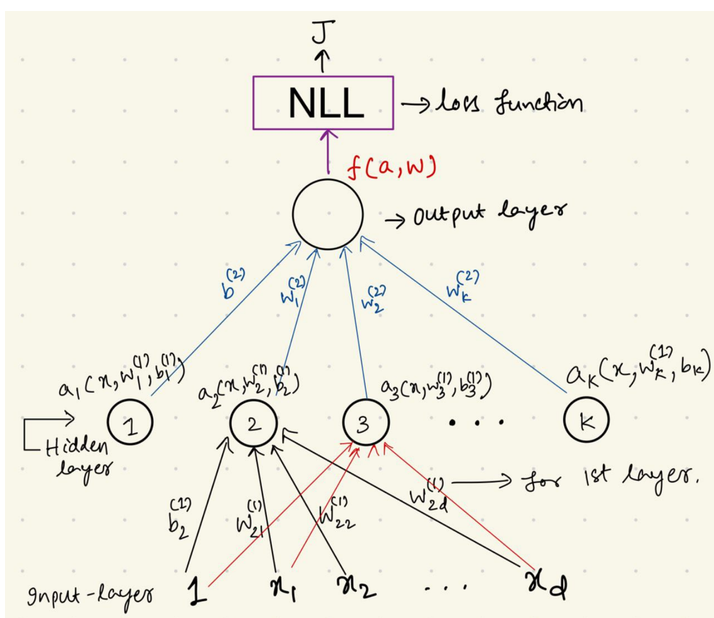


Figure 11.4: Multi-layer Neural Network

11.4 Popular Activation Functions

The activation function is analogous to the build-up of electrical potential in biological neurons, which then fire once a certain activation potential is reached. This activation potential is mimicked in artificial neural networks using probability.
The activation function should do two things:-
1. Ensure non-linearity to capture complex features that are not linear.
2. Ensure gradients remain large through the hidden layers in Deep Neural Networks; otherwise, if there are multiple layers, then the gradient will vanish, which is known as **vanishing gradient problem**.
Following are some of the popular activation functions:-

1. **Sigmoid** (σ) :-
$$\sigma(x) = \frac{1}{1+e^{-x}}$$
2. **Hyperbolic tan** (tanh) :-
$$\tanh(x) = \frac{e^{2x}-1}{e^{2x}+1}$$
3. **Rectified Linear Unit** (ReLU) :-
$$\text{ReLU}(x) = \max\{0, x\}$$

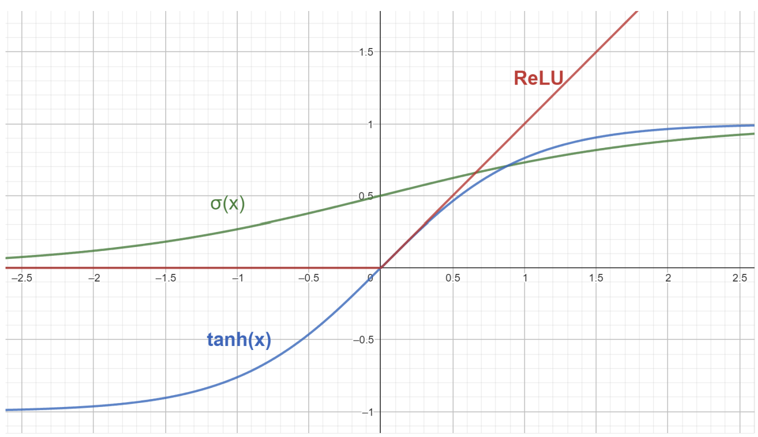


Figure 11.5: Activation functions' graphs

σ and tanh suffer from **vanishing gradient problem**, which is not a problem in the case of ReLU as the gradient does not vanish; that is why we generally use ReLU in hidden layers to avoid vanishing gradient.

11.5 Finding parameters

Find the values of w's and b's that minimize the loss function; this process is called the Training of Neural Networks. Let's discuss this in Feedforward Neural Network(FNN).
Feedforward Neural Network: It is one of the broad types of Artificial Neural Networks, where the flow of information is unidirectional and is from the inputs to outputs through hidden layers without any cycles or loops, in contrast to recurrent neural networks, which have a bi-directional flow.
For a full-blown Neural Network with dataset D= $(x_i, y_i)_{i=1}^{n}$,

Step-1: Define a loss function J(w,b) (i.e, J(θ))
Step-2: If f(x_i,θ) = NN($\mathbf{x_i}$,θ) Then,

$$J(\theta) = \Sigma_{i=1}^{n}\ell(NN(x_i,\theta), y_i)$$

If J is the Cross-Entropy Loss function, then

$$\ell(NN(x_i,\theta), y_i) = -(y_i log(NN(x_i,\theta)) + (1-y_i)log(1 - NN(x_i,\theta)))$$

This is for binary classification , where f(x_i,θ) is P(y_i=1 | x_i, θ). For multi-class classification we use softmax(x,θ)
After defining loss function, we use SGD(Stochastic Gradient Descent) to optimize.

11.6 NN training algorithm

- Inputs : NN(x,θ), training examples($x_1, x_2, ...x_n$), labels($y_1, y_2, ...y_n$) and loss function ℓ
- Randomly initialize θ
- Do until stopping criteria is met

Pick a random datapoint (x_i, y_i)

Compute gradient of ℓ at (x_i, y_i) , i.e, $\nabla_\theta\ \ell(x_i, y_i)$

$$\theta_{t+1} = \theta_t - \eta\nabla_\theta\ell$$

Return θ_{t+1}

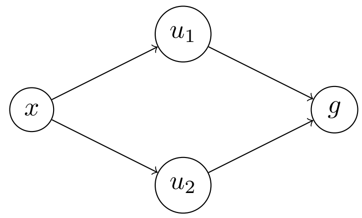
The data points can be very large, so to include the maximum number of points contribution in SGD, we require many iterations normally, hence we use Mini Batch Gradient Descent,this reduces the number of iterations
B $\subseteq$ {1,2,...,n},Here the update step is,

$$\theta_{t+1} = \theta_t - \eta\Sigma_{i\in B}\nabla_\theta\ell(x_i, y_i)$$

11.7 Backpropagation

The whole training of neural networks lies in the fact that how you can train it using backpropagation. It simply means that if you have loss l and θ is set of all parameters, then how will you calculate $\frac{d(l)}{d(\theta)}$ so that you can train by doing $\theta_1 -= \frac{d(l)}{d(\theta_1)}$ × learning_rate.

But my model is complex, so what I'll do is I'll divide the whole stuff into layers and derivatives of one layer with respect to the previous layer, then simply I'll use the chain rule. For example, if I want to calculate $\frac{dl}{da}$ where $f(a) = b$, $g(b) = l$, instead of directly $\frac{dl}{da}$, first db/da then dl/db, then $\frac{db}{da} \times \frac{dl}{db} = \frac{dl}{da}$.



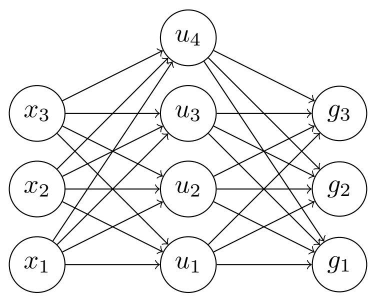
As layers will be vectors, let's move to vectors. Say x connected to u_1, u_2 connected to g, then

$$\frac{dg}{dx} = \frac{du_1}{dx} \times \frac{dg}{du_1} + \frac{du_2}{dx} \times \frac{dg}{du_2}$$

$$\frac{dg}{dx} = \frac{\partial \vec{u}}{\partial x} \cdot \frac{\partial g}{\partial \vec{u}}$$

Where $\frac{\partial \vec{u}}{\partial x} = (\frac{\partial u_1}{\partial x}, \frac{\partial u_2}{\partial x})$ and $\frac{\partial g}{\partial \vec{u}} = (\frac{\partial g}{\partial u_1}, \frac{\partial g}{\partial u_2})$

Now let's move to multiple vectors



In a multi-layer neural network, each layer consists of nodes and connections between nodes carry weights. During both forward pass (computing the output) and backward pass (computing gradients for training), derivatives play a crucial role. Here, we discuss the computation of derivatives with respect to inputs and the matrix representation of these derivatives.

The derivative $\frac{\partial \vec{u}}{\partial \vec{x}}$ represents the sensitivity of the intermediate layer u to changes in the input x. It can be represented as a matrix where each element (i, j) corresponds to the partial derivative of u_i with respect to x_j. This matrix is commonly known as the Jacobian matrix.

$$\frac{\partial \vec{u}}{\partial \vec{x}} = \begin{bmatrix} \frac{\partial u_1}{\partial x_1} & \frac{\partial u_2}{\partial x_1} & \cdots & \frac{\partial u_k}{\partial x_1} \\ \frac{\partial u_1}{\partial x_2} & \frac{\partial u_2}{\partial x_2} & \cdots & \frac{\partial u_k}{\partial x_2} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial u_1}{\partial x_d} & \frac{\partial u_2}{\partial x_d} & \cdots & \frac{\partial u_k}{\partial x_d} \end{bmatrix}$$

Similarly, for the matrix of $\frac{\partial \vec{g}}{\partial \vec{u}}$, it would have elements representing the partial derivatives of each element in $\vec{g}$ with respect to each element in $\vec{u}$.

$$\frac{\partial \vec{g}}{\partial \vec{u}} = \begin{bmatrix} \frac{\partial g_1}{\partial u_1} & \frac{\partial g_2}{\partial u_1} & \cdots & \frac{\partial g_m}{\partial u_1} \\ \frac{\partial g_1}{\partial u_2} & \frac{\partial g_2}{\partial u_2} & \cdots & \frac{\partial g_m}{\partial u_2} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial g_1}{\partial u_k} & \frac{\partial g_2}{\partial u_k} & \cdots & \frac{\partial g_m}{\partial u_k} \end{bmatrix}$$

Consider vectors $\vec{x}$, $\vec{u}$, and $\vec{g}$, where every node in $\vec{x}$ is connected to every node in $\vec{u}$, and every node in $\vec{u}$ is connected to every node in $\vec{g}$. The derivative $\frac{\partial g_i}{\partial x_j}$ represents the sensitivity of each element g_i in $\vec{g}$ to changes in each element x_j in $\vec{x}$. This derivative can be computed using the chain rule:

$$\frac{\partial \vec{g}}{\partial \vec{x}} = \frac{\partial \vec{u}}{\partial \vec{x}} \cdot \frac{\partial \vec{g}}{\partial \vec{u}}$$

$$\frac{\partial g_i}{\partial x_j} = \sum_{z=1}^{k} \left(\frac{\partial u_z}{\partial x_j} \cdot \frac{\partial g_i}{\partial u_z} \right)$$

Here, we sum over all elements u_z, where each term in the sum involves the product of two partial derivatives.

References

- Medium article - https://towardsdatascience.com/simple-introduction-to-neural-networks-ac1d7c3d7a2
- History of Neural Networks - https://cs.stanford.edu/people/eroberts/courses/soco/projects/neural-networks/History/history2.html