# IDC410

# A course on Image Processing and Machine Learning

## (Lecture 10)

**Shashikant Dugad,**

**IISER Mohali**

# Reading Material

## Suggested Books:

1. Neural Networks and Deep Learning by Michael Nielsen
2. Fundamentals of Deep Learning by Nikhil Buduma

## Source for this presentation:

**Neural Networks and Deep Learning by Michael Nielsen**

https://www.scaler.com/topics/deep-learning/introduction-to-feed-forward-neural-network/

https://www.turing.com/kb/mathematical-formulation-of-feed-forward-neural-network

http://machine-learning-for-physicists.org. by Florian Marquardt

3Blue1Brown (Youtube Videos)

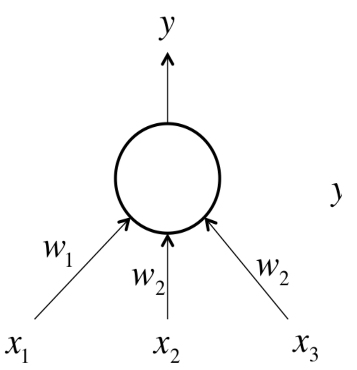
https://www.datacamp.com/tutorial/introduction-to-activation-functions-in-neural-networks

https://www.geeksforgeeks.org/introduction-to-recurrent-neural-network/

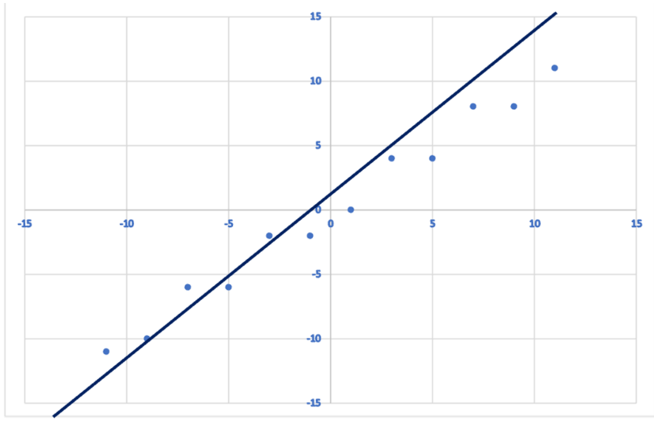
# Network Training

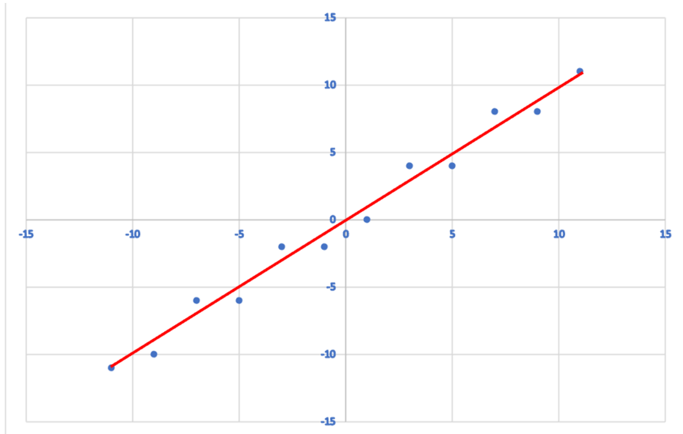
# Training Feed-Forward Neural (FFN) Networks

- Let us try to understand the concepts of training using a simple neural network with **linear activation function** (**output $y^i$== input**) i.e., $y^i = w_1 x_1^i + w_2 x_2^i + w_3 x_3^i$
- **Known parameters during training**: # of Idli, Parantha, Lassi $(x_1^i, x_2^i, x_3^i)$ purchased and total cost (***$t^i$***) associated with the **i**$^{th}$ purchase
- Let us assume purchases of these items were done large # of times (***i >> 1***)
- **Task:** Determine the true cost of each item (***$w_1$, $w_2$*** and ***$w_3$***)




**Idli Parantha Lassi**



# Training Feed-Forward Neural (FFN) Networks

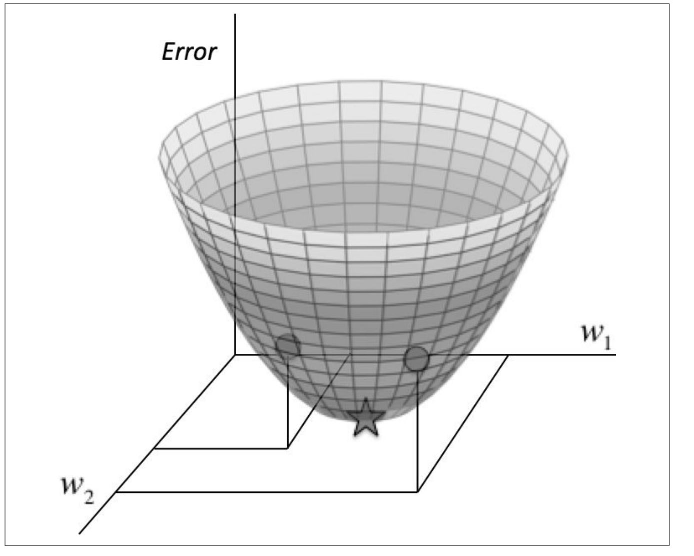
- If $t_i$ is the true total cost (known parameter) and $y_i$ is the total cost predicted by the network for $i^{th}$ purchase for a given value of weights in that iteration; then, we want to tune each of these weights in such a way that difference between predicted and true cost ($|t_i - y_i|$) is as small as possible!
- We shall choose those weights as our final weights, when the overall difference between the predicted and true cost is **smallest** across all the purchases. The **Loss Function** is defined as:

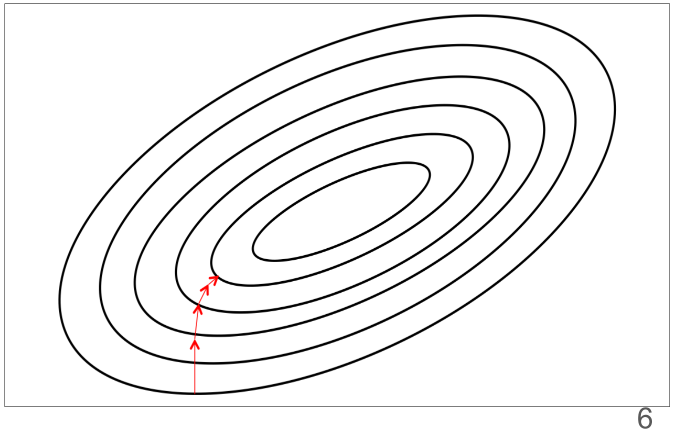
$$E = \frac{1}{2}\sum_{i=1}^{n}\left(t^i - y^i\right)^2$$

- As a result, our goal will be to select our parameter vector **θ** (the values for all the weights in our model) such that, the **loss function E** is as close to **0** as possible. Note: *E* is always *>0*
- This problem can be easily solved in exact manner with three purchases of different types
- However, situation changes completely with non-linear activation function

# Gradient Descent

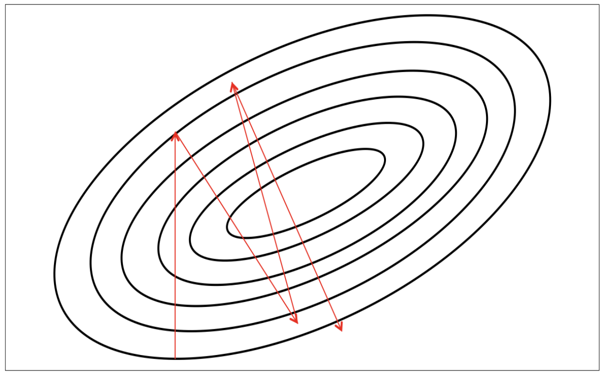
- To understand the importance of gradient, let us simplify the previous problem by assigning only two (instead of three) inputs to our linear neuron with weights $w_1$ and $w_2$.
- Imagine a 3-dimensional space where the horizontal dimensions correspond to the weights $w_1$ and $w_2$ and the vertical dimension corresponds to the ***error (loss) function*** $E(w_1, w_2)$.
- Shape of the error function for all possible weights would be more like a quadratic bowl
- We can visualize this surface as a set of elliptical contours with minimum error at the center of the ellipses.
- Closer the contours to each other, the steeper will be the slope. Direction of the steepest descent is always perpendicular to the contours.
- This direction is expressed as a vector known as the ***Gradient***.



# Training: Delta Rule and Learning Rates (LR)

- Parameters to be determined (***$w_1$*** and ***$w_2$***) by way of training are referred as **hyperparameters**.
- The training algorithm needs additional hyperparameters such as **learning rate**.
- Magnitude of a step of a walk along direction of gradient of steepest descent is determined by the **learning rate (*LR*)**
- The step size (learning rate) will be driven by the steepness of the surface.
- Closer we are to the minimum (flat gradient), the shorter we want to step forward
- Mellow (flatter) shape of the **error surface** may take large training time
- The step size is determined by multiplying the gradient with the learning rate ($\Delta\omega_k = \varepsilon G_k$)
- ***Small LR*** may result in better accuracy with large training time. ***Higher LR*** may have difficulty in converging to minima
- Training algorithms can be built with an ***adaptive LR*** to automate the process of dynamically selecting the ***LR***



# Training: Delta Rule and Learning Rate

- Change in weight (***$\Delta\omega_k$***) while taking next step, is evaluated using the LR (**ε**) and the gradient (***$G_k$***). The gradient is a partial derivative of the ***error function*** (***$E(\omega_k)$***) with respect to each of the weight

$$G_k = \frac{\partial E}{\partial \omega_k} = \frac{\partial}{\partial \omega_k}\left(\frac{1}{2n}\sum_{i=1}^{n}\left(t^i - y^i\right)^2\right)$$

$$\therefore\ G_k = -\left(\frac{1}{n}\sum_{i=1}^{n}(t_i - y_i)\frac{\partial y^i}{\partial \omega_k}\right) = -\frac{1}{n}\sum_{i=1}^{n} x_k^i\left(t^i - y^i\right)$$

$$\boxed{\therefore\quad \Delta\omega_k = -\epsilon G_k = \epsilon\frac{1}{n}\sum_{i=1}^{n} x_k^i\left(t^i - y^i\right)}$$

- Please note the index ***i*** denotes number of purchases made. ***$t^i$*** is the actual (true) bill and ***$y^i$*** is the estimated bill.

# Training with nonlinear activation function

- Let us now consider a neuron with the sigmoidal activation function. It's a non-linear activation function. The output of this neuron will be:

$$y = \sigma(z) = \frac{1}{1 - e^{-z}} \quad where, \qquad z = \sum_{k=1}^{m} \omega_k \, x_k + b = \sum_{k=0}^{m} \omega_k \, x_k$$

- Note, m is number of neurons, $w_0$=$b$ and $x_0$=$1$
- The neuron computes the weighted sum of its inputs (the logit z) and then feeds its logit into the input function to compute it final output y
- Sigmoid has very nice derivatives, which makes learning easy! Let us get the gradient now

$$\frac{\partial z}{\partial \omega_k} = x_k, \qquad \frac{\partial z}{\partial x_k} = \omega_k \qquad \frac{\partial y}{\partial z} = y(1-y)$$

$$\frac{\partial y}{\partial \omega_k} = \frac{\partial y}{\partial z}\frac{\partial z}{\partial \omega_k} = x_k y(1-y)$$

# Training with nonlinear activation function

- **Compute the derivative of the error function with respect to each weight:**

$$G_k = \frac{\partial E}{\partial \omega_k} = \frac{\partial \sum_{i=1}^{n} \left[\frac{1}{2}\left(t^i - y^i\right)^2\right]}{\partial \omega_k} = \sum_{i=1}^{n} \frac{\partial \frac{1}{2}\left(t^i - y^i\right)^2}{\partial y^i} \frac{\partial y^i}{\partial \omega_k} = -\sum_{i=1}^{n} \left(t^i - y^i\right) x_k^i \, y^i\left(1 - y^i\right)$$

$$\boxed{\Delta\omega_k = -\epsilon G_k = \sum_{i=1}^{n} \epsilon \, x_k^i y^i\left(1 - y^i\right)\left(t^i - y^i\right)}$$

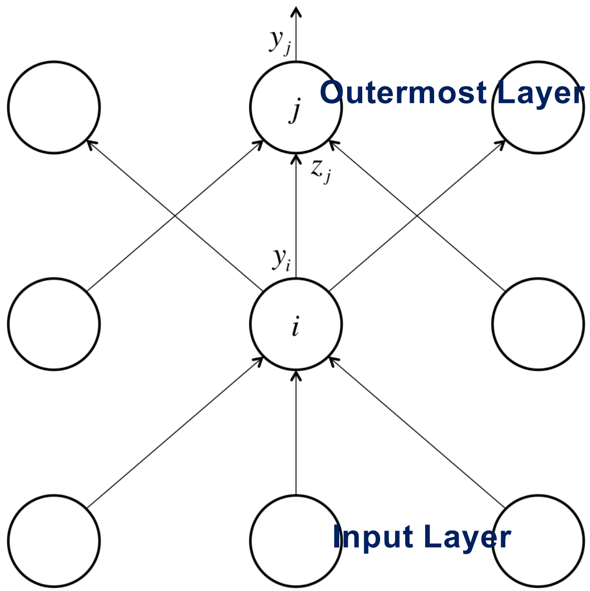
- **Note:** the index ***i*** denotes number of events (purchase), n denotes total no. of purchases, ***k*** denotes the neuron number at input layer. $t^i$ is the actual, i.e., true output (bill) and $y^i$ is the estimated (predicted) output (bill)

- This completes the formalism for evaluating weights for ***m*** **inputs connected to the** ***ONE*** **neuron in the output layer with** ***NO hidden layers*** in-between

# Backpropagation Algorithm (Training Multi Layer Network)

- Now, we consider the case of multi-neuron output layer with a multi-neuron hidden layer between the input and the output layer
- Backpropagation algorithm for muti-layer FFN was developed by David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams in 1986
  - Rumelhart, David E., Geoffrey E. Hinton, and Ronald J. Williams. "Learning representations by backpropagating errors." Cognitive Modeling 5.3 (1988)
- Backpropagation has to deal with extremely high dimensional space.
- ***j*** and ***i*** refers to the neuron number of current and the preceding layer #, ***y*** to the activation output and $z_j$ to the logit of the neuron ***j***
- Assuming we know the error derivatives of layer ***j***, we can use it compute the same for the previous layer ***i***
- The error function derivatives at the output layer:

$$E = \frac{1}{2} \sum_{j \in output} \left(t_j - y_j\right)^2 \Rightarrow \frac{\partial E}{\partial y_j} = -\left(t_j - y_j\right)$$



# Backpropagation Algorithm (Training Multi Layer Network)

- We can now determine how the error changes w. r. t. the weights. This gives us how to modify the weights after each training example:

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial z_j}{\partial w_{ij}} \frac{\partial y_j}{\partial z_j} \frac{\partial E}{\partial y_j} \qquad \frac{\partial y_j}{\partial z_j} = y_j(1 - y_j) \qquad \frac{\partial z_j}{\partial w_{ij}} = y_i$$

- Therefore,

$$\therefore \frac{\partial E}{\partial w_{ij}} = y_i y_j (1 - y_j) \frac{\partial E}{\partial y_j}$$

- Finally, to complete the algorithm, just as before, we merely sum up the partial derivatives over all the training examples in our designated training dataset. This gives us the following modification formula:

$$\boxed{\Delta w_{ij} = -\sum_{k \in Dataset} \epsilon y_i^k y_j^k (1 - y_j^k) \frac{\partial E^k}{\partial y_j^k}}$$

# Backpropagation Algorithm (Training Multi Layer Network)

- Note: $z_j = w_{ij} y_i + b_j \quad \Rightarrow \quad \frac{\partial z_j}{\partial y_i} = w_{ij}$
- **Note:** Change in a single neuron output (***y**$_i$*) in the middle layer impacts logits of every neuron in the outermost layer.
- Impact of change in $y_i$ on $E$ can be obtained, using the fact that the partial derivative of the logit w.r.t. the output data ($y_i$) from the layer $i$, is merely the weight of the connection $w_{ij}$:

$$\frac{\partial E}{\partial y_i} = \sum_{j \in output} \frac{\partial E(y_j)}{\partial y_i} = \sum_{j \in output} \frac{\partial E}{\partial y_j} \frac{\partial y_j}{\partial z_j} \frac{\partial z_j}{\partial y_i} = \sum_{j \in output} w_{ij} y_j (1 - y_j) \frac{\partial E}{\partial y_j}$$

- **Note:** Layer $h$ precedes Layer $i$ and Layer $i$ is precedes Layer $j$

$$\Delta w_{hi} = - \sum_{k \in Dataset} \epsilon y_h^k y_i^k (1 - y_i^k) \frac{\partial E^k}{\partial y_i^k} = - \sum_{k \in Dataset} \epsilon y_h^k y_i^k (1 - y_i^k) \sum_{j \in output} w_{ij} y_j (1 - y_j) \frac{\partial E}{\partial y_j}$$