Week1 – Biological neurons

- Reticular Theory (nervous system is a continuous network, rather than individual cells) proposed by Joseph von Gerlach – 1871. This was confirmed in the 1950's due to electron microscopy.
- Camillo Golgi discovered a chemical reaction that helped examine nervous tissue.
- Neuron doctrine was consolidated on 1891 by Heinrich Wilhem Gottfried, who also coined the term chromosome.
- McCulloch and Pitts proposed a simplified (binary) model of neuron in 1943.
- Perceptron was pitched by Frank Rosenblatt during 1958, and roughly is the same as the previous model, except that weights were added to each input.
- Earliest ancestor of DL networks was the multi-layer perceptrons (MLP) proposed by Ivakhnenko. Many problems that are unsolvable using single-neurons can be solved using networks of neurons. MLP with a single hidden layer can be used to approximate any continuous function to any desired precision
- Minsky and Papert outlined the limits of perceptrons, that triggered the AI winter for nearly two decades until late 1980's.
- Backpropagation was proposed in the 1960's, but got concretized in 1986 due to the paper written by Rumelhart, Geoffrey Hinton et al. Gradient descent (1847 - Cauchy) combined with backpropagation helped in laying the foundation.
- Work restarted on deep learning in 2006, when Hinton et al proposed an effective way to initializing weights in the layers of a network.
- Following summarizes the journey of deep neural networks over the years, through various architectures and proposed solutions.

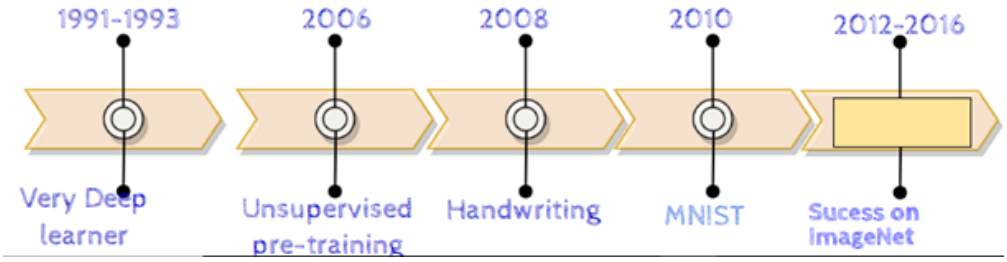


Figure 1: Deep neural networks over the decades.

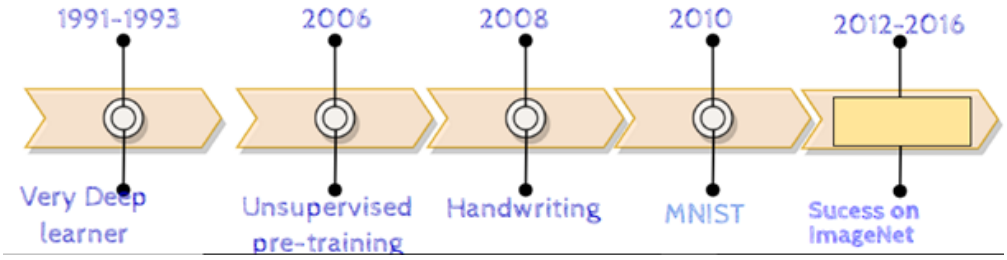


Figure 2: Evolution of convolutional neural networks.

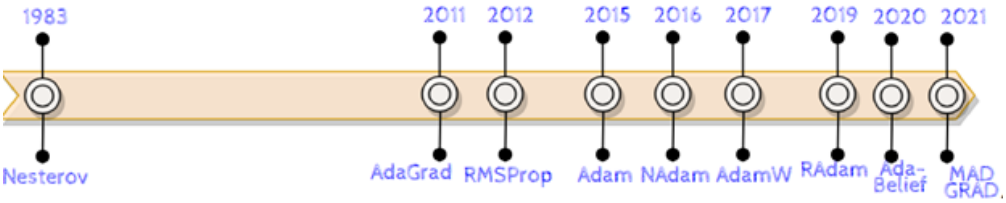


Figure 3: Faster convergence

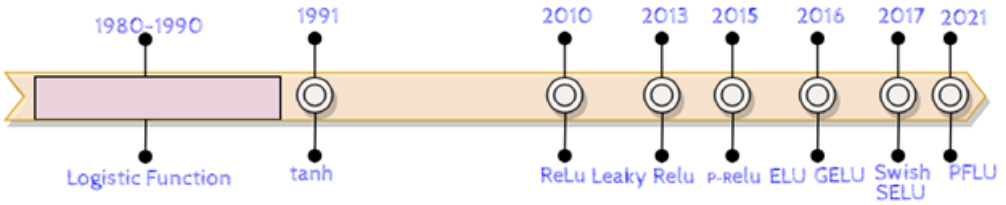


Figure 4: Better activation functions

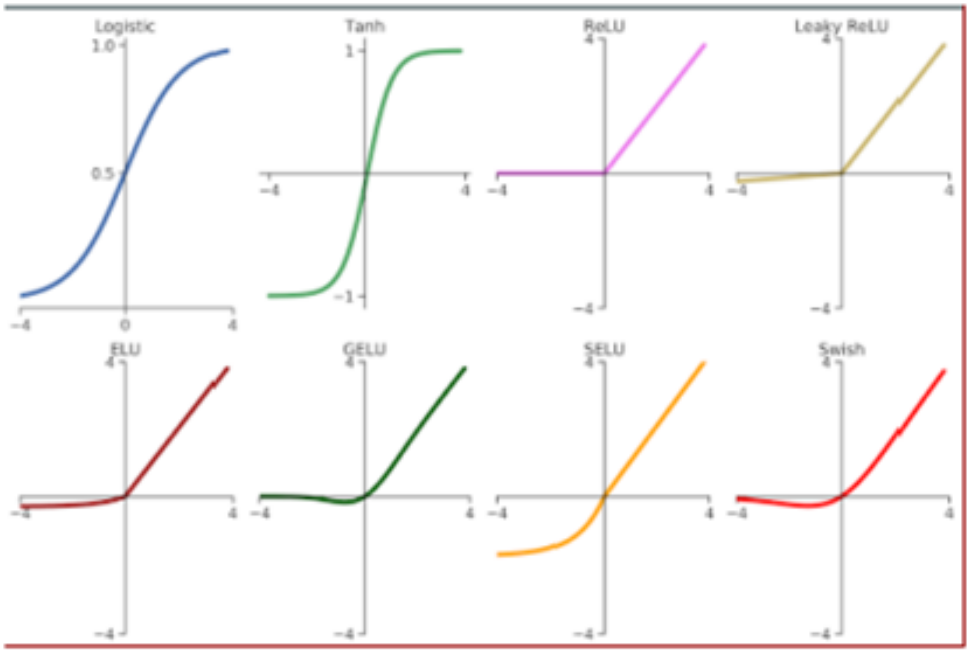


Figure 5: Various convergence functions

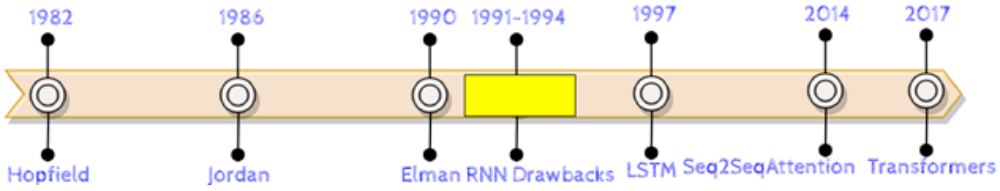


Figure 6: Sequences (sound/video)

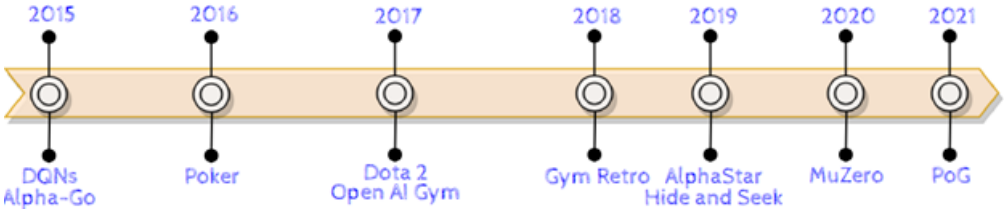


Figure 7: Games

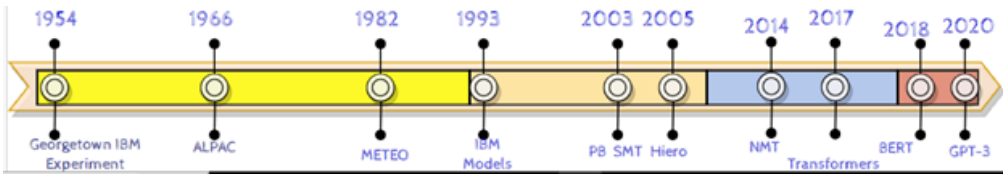


Figure 8: Rise of transformers

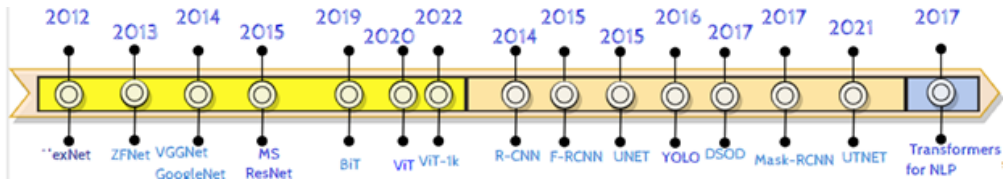


Figure 9: Transformers everywhere!

Legend: *Yellow* = Image classification, *Grey* = Object detection and segmentation, *Blue* = NLP.

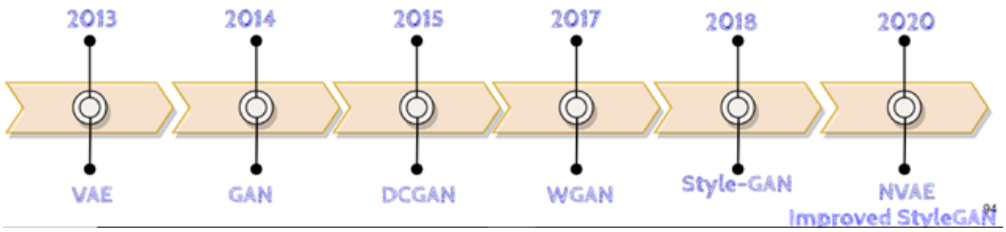


Figure 10: Generative models

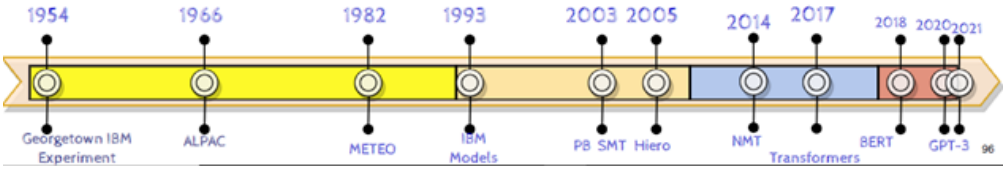


Figure 11: Language-Vision

- In spite of high capacity, deep learning is not susceptible to overfitting.
- Challenges of deep learning
  - Numerical instability (vanishing/exploding gradients)
  - Overfitting (sharp minima)
  - Not robust
- In a biological neuron,
  - dendrite receives signals from other neurons.
  - synapse helps connect to other neurons.
  - soma processes the information
  - axon transmits output of this neuron

- Neural network in the brain is a massively parallel, layered network and ensures there's division of work. Each neuron may perform a certain role or respond to a certain stimulus. Neurons in each layer gets activated depending on the output from the previous layers.
- McCulloch Pitts Neuron
  - It is a computational model that mimics the biological neuron. Inputs are boolean, and output is a boolean.

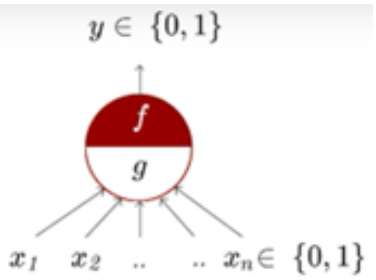


Figure 12: McCulloch Pitts Neuron

  - In the above figure, g is a function that aggregates (sums up) all inputs. f is a function that outputs a boolean based on the aggregation.
  - The inputs can be excitatory or inhibitory.
  - Thus,

$$y = 0 \; if \; any \; x_i \; is \; inhibitory, \; else$$
$$y = f(g(x)) = 1 \; if \; g(x) \geqslant \theta, \; and$$
$$y = f(g(x)) = 0 \; if \; g(x) < \theta$$
$$where \; g(x_1, x_2 ... x_n) = g(x) = \sum_{i=1}^{n} x_i \qquad (1)$$
$$NOTE: \; \theta \; is \; called \; the \; threshold.$$

  - In order to implement AND logic with 3 inputs x_1, x_2, x_3 the threshold should be set to 3.
  - Similarly, in order to implement OR logic with 3 inputs x_1, x_2, x_3 the threshold should be set to 1.
  - The neuron below represents the OR logic for 2 inputs.

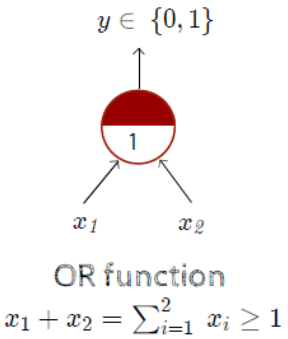


Figure 13: McCulloch Pitts neuron representing an OR logic with 2 inputs

The above neuron is geometrically represented as below.

Here, any point that lies above the line $x_1 + x_2 = 1$ is considered positive and any point that lies below is considered negative, thus essentially dividing all inputs into two different $half$ spaces.

– Similarly, for a neuron that represents the OR logic for 3 inputs (shown below)

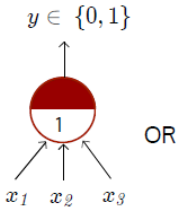
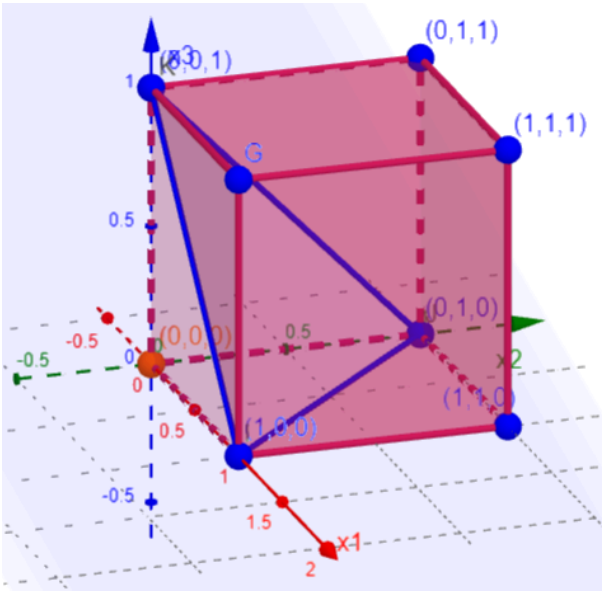


Figure 14: Neuron that represents OR logic with 3 inputs

The above neuron is geometrically represented as below.



Here, any point that lies above the plane $x_1 + x_2 + x_3 = 1$ is considered positive and any point that lies below is considered negative, thus essentially dividing all inputs into two different $half$ spaces.

- In all above cases, notice that the intercept of the hyperplane that divides positive and negative half-spaces (decision boundary) is given by the bias. When the bias is 0, the hyperplane will pass through the origin.
- Bias (threshold) of a MP neuron can be obtained using the formula $\theta = nw - p$, where n is the number of inputs, w is the positive weight and p in the number of inhibitory inputs.

Problems

- Decision line is given by $w^T x = 0$, all points that are $w^T x > 0$ gets classified as class-1, and all $w^T x < 0$ gets classified as class-2. If the weight vector is $[-0.5, -0.5]$. in the graphical representation will class-1 appear above the line or below the line?

  For a weight vector is $[-0.5, -0.5]$, then the equation $w^T x$ can be written as $-0.5x_1 - 0.5x_2 = 0$, which can be rewritten as $x_2 = -x_1$ or the more familiar form for line's equation $y = -x$. This is a line with a slope of -1 and passing through $(0, 0)$. Points in class-1 belong to

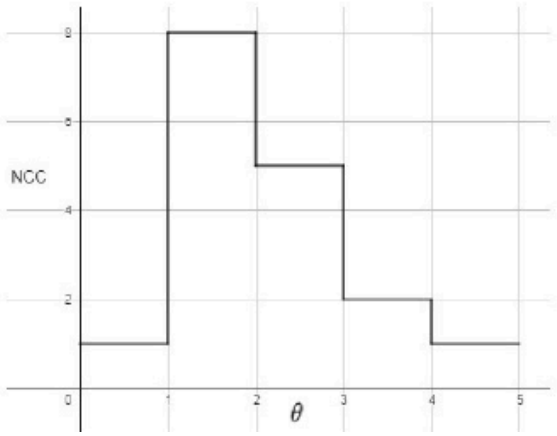
$$w^T x > 0 \implies -0.5x_1 - 0.5x_2 > 0 \implies -0.5x_1 > 0.5x_2$$

Dividing both sides by 0.5, we get $-x_1 > x_2 \implies x_2 < -x_1$. This implies that all positive points are below the line $x_2 = -x_1$. Similarly, points in class-2 are above the line $x_2 = -x_1$

- https://discourse.onlinedegree.iitm.ac.in/t/jan-23-q-108-cannot-understand-how-this-could-be-possible/138754

Suppose that we implement a three input Boolean function using the Mc-Culloch Pitts (MP) neuron. The graph below shows the Number of Correctly Classified (NCC) data points for various values of threshold θ. The threshold θ is incremented by 1 from 0 to 5. Assume that the neuron does not have any inhibitory input. This graph represents which of the following Boolean functions?

$$\hat{y} = \begin{cases} 1, & \text{if } \sum x_i \geq \theta \\ 0, & \text{otherwise} \end{cases}$$



Answer: Above graph represents an OR function.
Explanation: See the following table.

x1	x2	x2	Sum	x=0	x=1	x=2	x=3	x=4	x=5	OR
0	0	0	0	>=0	<1	<2	<3	<4	<5	0
0	0	1	1	>=0	>=1	<2	<3	<4	<5	1
0	1	0	1	>=0	>=1	<2	<3	<4	<5	1
0	1	1	2	>=0	>=1	>=2	<3	<4	<5	1
1	0	0	1	>=0	>=1	<2	<3	<4	<5	1
1	0	1	2	>=0	>=1	>=2	<3	<4	<5	1
1	1	0	2	>=0	>=1	>=2	<3	<4	<5	1
1	1	1	3	>=0	>=1	>=2	>=3	<4	<5	1
Green indicates the number of correctly classified points.										
Orange indicates the number of incorrectly classified points.										

NOTE: For $x = 0$, the graph should have started with $NCC = 7$. This seems to be a mistake in the paper.

- Perceptrons Vs McCulloch Pitts neuron
  - Perceptrons can handle non-boolean inputs, whereas McCulloch Pitts could handle only boolean inputs.
  - Perceptrons allow weights to be specified for each input. It's possible to learn the

weights due to optimization algorithms, instead of hand coding them.

- Mathematical representation of a Perceptron

$$y = 1 \; if \sum_{i=1}^{n} w_i * x_i \geqslant \theta$$
$$= 0 \; if \sum_{i=1}^{n} w_i * x_i < \theta$$

$$OR$$

$$y = 1 \; if \sum_{i=1}^{n} w_i * x_i - \theta \geqslant 0$$
$$= 0 \; if \sum_{i=1}^{n} w_i * x_i - \theta < 0$$

- Above set of equations could be rewritten as

$$y = 1 \; if \sum_{i=0}^{n} w_i * x_i \geqslant 0 \tag{2}$$
$$= 0 \; if \sum_{i=0}^{n} w_i * x_i < 0$$
$$where \; x_0 = 1 \; and \; w_0 = -\theta$$

- w_0 is termed the *bias*, because rest of the inputs or their corresponding weights should be high enough to cross 0, inspite of a high negative value (prejudice/bias) due to w_0
- Essentially, a perceptron will fire if the weighted sum of its inputs is greater than the threshold (-w_0)
- Perceptron, like McCulloch Pitts, separates input space into two halves. But, we've a learning algorithm to learn the weights.
- In the case of OR logic, following table represents the input/output from a perceptron.

x_1	x_2	OR	
0	0	0	$w_0 + \sum_{i=1}^{2} w_i x_i < 0$
1	0	1	$w_0 + \sum_{i=1}^{2} w_i x_i \geq 0$
0	1	1	$w_0 + \sum_{i=1}^{2} w_i x_i \geq 0$
1	1	1	$w_0 + \sum_{i=1}^{2} w_i x_i \geq 0$

- One of the solutions for the above set of equations is $w_0 = -1,\ w_1 = 1.1,\ w_2 = 1.1.$ If we construct a line with these weights, all the above 4 inputs get classified properly, implying $error = 0$ in this case.
- Note that if the inequality evaluates to $\geqslant 0$, then it's called a positive half space, else it's called the negative half space.
- Following are the error computations in a perceptron that implements AND function, for different weight values.

For w0 = -1, w1 = -1, w2 = -1,

x1	x2	w0+w1x1+w2x2	Act. output	Exp.output (OR function)
0	0	-1	0	0
0	1	-2	0	0
1	0	-2	0	0
1	1	-3	0	1

This results in error = 1, since actual and expected outputs don't match for one input

For w0 = -1, w1 = 1.5, w2 = 0,

x1	x2	w0+w1x1+w2x2	Act. output	Exp.output (OR function)
0	0	-1	0	0
0	1	-1	0	0
1	0	0.5	1	0
1	1	0.5	1	1

This results in error = 1, since actual and expected outputs don't match for one input

For w0 = -1, w1 = 10, w2 = -10,

x1	x2	w0+w1x1+w2x2	Act. output	Exp.output (OR function)
0	0	-1	0	0
0	1	-11	0	0
1	0	9	1	0
1	1	-1	0	1

This results in error = 2, since actual and expected outputs don't match for two inputs

Figure 15: Error computations in a perceptron

Note that the maximum error is 3 (and not 4), since any one out of the four points will be correctly classified at all times.

- Algorithm for perceptron learning - pseudo code.

$P \leftarrow inputs\ with\ label\ 1;$
$N \leftarrow inputs\ with\ label\ 0;$
Initialize $\mathbf{w}$ randomly;
while $!convergence$ **do**
  Pick random $\mathbf{x} \in P \cup N$;
  if $\mathbf{x} \in \mathrm{P}$ and $\sum_{i=0}^{n} w_i * x_i < 0$ **then**
    $\mathbf{w} = \mathbf{w} + \mathbf{x}$;
  end
  if $\mathbf{x} \in \mathrm{N}$ and $\sum_{i=0}^{n} w_i * x_i \geq 0$ **then**
    $\mathbf{w} = \mathbf{w} - \mathbf{x}$;
  end
end

Figure 16: Perceptron learning algorithm

- Let's see how we arrived at this algorithm, specifically the weight update. Note that eq. (2) be rewritten as

$$y = 1\ if\ w^T x \geqslant 0 \quad (3)$$
$$= 0\ if\ w^T x < 0$$
$$where\ w = [w_0,\ w_1,\ w_2,\ ...,\ w_n]\ and\ x = [1,\ x_1,\ x_2,\ ...,\ xn]$$

NOTE: $w^T x$ is also called the dot product of w with x.

- This can be diagrammatically represented as follows

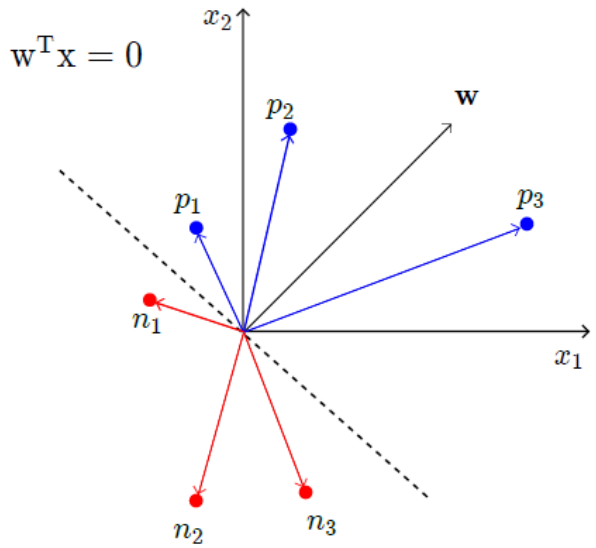


Figure 17: Weight vector casts $90°$ with data points on the dividing line $w^T x = 0$

- From the above diagram, it follows that,

$$Cos\alpha \;=\; \frac{w^T x}{||w||\;||x||} \tag{4}$$

- From eq.(4), it follows that all points x that lies on the line $w^T x \;=\; 0$ are at a right angle to vector w. This is because $Cos\alpha \;=\; 0$ for these points, which in turn implies that $\alpha \;=\; 90°$
- For all points $p_1,\; p_2,\; p_3$ in the positive half space, the angle cast by the weight vector with these points is less than $90°$. For all points $n_1,\; n_2,\; n_3$ in the negative half space, the angle cast by the weight vector with these points is more than $90°$.
- Now, with every update $w_{new} \;=\; w + x$,

$$Cos(\alpha_{new}) \;\propto\; \left((w_{new})^T x\right) \;\propto\; (w \;+\; x)^T x \;\propto\; \left(w^T x \;+\; x^T x\right) \;\propto\; \left(Cos\alpha \;+\; x^T x\right)$$
$$\implies Cos(\alpha_{new}) > Cos\alpha \implies \alpha_{new} < \alpha$$

  Thus, the angle between w and x gets reduced.
- Similarly, with every update $w_{new} \;=\; w - x$, $\alpha_{new} \;>\; \alpha$
- During each iteration, the weight vector gets recalculated. The algorithm will run until all input vectors in the set are in the correct half-space of the weight vector. This is called the convergence.
- Note that the update algorithm can be written as

$$w = \begin{cases} w \;-\; x, & if\; w^T x \;\geqslant\; 0,\; x \;\in\; N \\ w \;+\; x, & if\; w^T x \;<\; 0,\; x \;\in\; P \end{cases}$$

  or alternatively as

$$w = w + (y - \widehat{y})x$$

- During the process of weight updates, the angle between weights and individual data points could increase or decrease. However, note that the angle between the weight vector and the decision boundary always remain $90°$
- For linearly separable data, the learning is guaranteed to converge.

Problems

- In order to create a perceptron that classifies all inputs of a NAND gate given below,

x_1	x_2	y
0	0	1
1	0	1
0	1	1
1	1	0

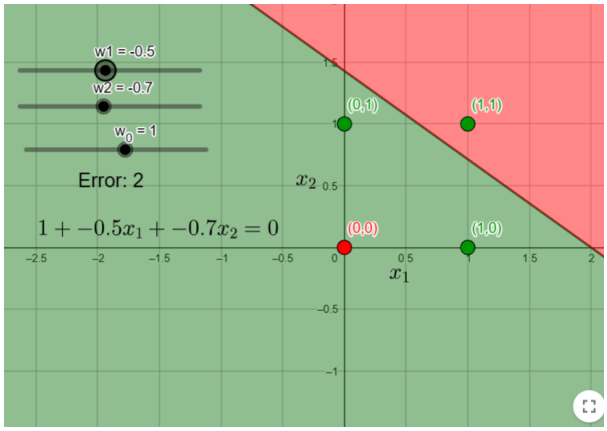
Figure 18: NAND truth table

we will write the inequalities for each of the inputs above as follows:

$$w_0 + w_1.0 + w_2.0 > 0 \implies w_0 > 0$$
$$w_0 + w_1.1 + w_2.0 > 0 \implies w_0 + w_1 > 0$$
$$w_0 + w_1.0 + w_2.1 > 0 \implies w_0 + w_2 > 0$$
$$w_0 + w_1.1 + w_2.1 < 0 \implies w_0 + w_1 + w_2 < 0$$

One of the possible set of values for the weights are:
$w_0 = 1,\ w_1 = -0.5,\ w_2 = -0.7$

Here is the geometric representation of the line that divides the points into positive and negative half-spaces.



- 2. (3 points) Suppose we have a perceptron with two inputs, x_1 and x_2. This perceptron undergoes training on a small dataset containing three points: $(-1, 2)$ labeled as class 0, $(0, -1)$ labeled as class 1, and $(2, 1)$ labeled as class 0. The weights of the perceptron are initialized to zeros, and the model is trained until it reaches convergence. Given this scenario, what would be the assigned output class by the trained perceptron for the new point $(-2, 0)$? Ignore the bias term in the model.

Solution:

x	w	xw	xw >=0	y_hat	update
(-1,2)	(0,0)	0	1	0	w = w - x = (0,0) - (-1,2) = (1,-2)
(0,-1)	(1, -2)	2	1	1	no update
(2,1)	(1,-2)	0	1	0	w = w - x = (1,-2) - (2,1) = (-1,-3)
(-1,2)	(-1,-3)	-5	0	0	no update
(0,-1)	(-1,-3)	3	1	1	no update
(2,1)	(-1,-3)	-5	0	0	no update
Final weight vector w = (-1,-3)					
For x = (-2,0), xw = 2, which is >=0 and hence classified as 1.					

Week2

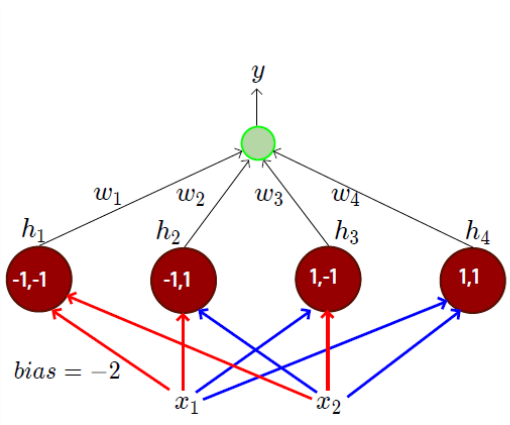
- From a 2 input perceptron, one could generate 16 different functions. Of these, XOR

and !XOR function are non-linearly separable functions. All others are linearly separable.

x_1	x_2	f_1	f_2	f_3	f_4	f_5	f_6	f_7	f_8	f_9	f_{10}	f_{11}	f_{12}	f_{13}	f_{14}	f_{15}	f_{16}
0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
1	0	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
0	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

Figure 19: Functions from a 2-input perceptron

- In general, the formula for the possible functions from a perceptron with n inputs is 2^{2^n}. The number of non-linearly separable functions for an n-input perceptron is not known. It's an unsolved problem.
- A single perceptron is unable to handle non-linearly separable functions. However, a network of perceptrons can.
- To start with, we will consider a single hidden layer apart from the input and output layers. Given n inputs in the input layer, the hidden layer should've 2^n perceptrons, each of which is responsible for one input combination and one output layer containing 1 perceptron.



Let w_0 be the bias output of the neuron (i.e., it will fire if $\sum_{i=1}^{4} w_i h_i \geq w_0$

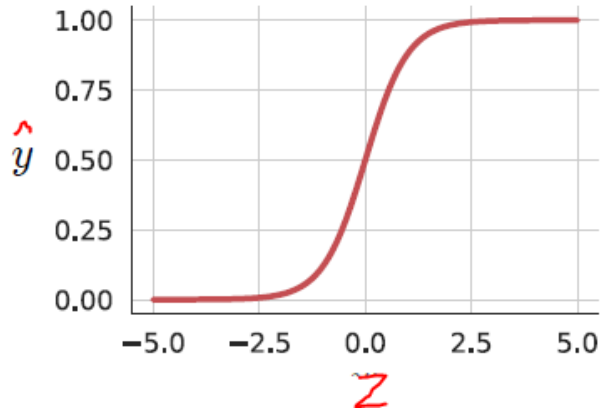
x_1	x_2	XOR	h_1	h_2	h_3	h_4	$\sum_{i=1}^{4} w_i h_i$
0	0	0	1	0	0	0	w_1
0	1	1	0	1	0	0	w_2
1	0	1	0	0	1	0	w_3
1	1	0	0	0	1	1	w_4

This results in the following four conditions to implement XOR: $w_1 < w_0$, $w_2 \geq w_0$, $w_3 \geq w_0$, $w_4 < w_0$

Figure 20: Network of perceptrons can solve all input combinations

- A hidden layer of m neurons is capable of implementing 2^m functions.
- The issue is that for a large value of n, the MLP would be unmanageable. As an example, creating a hidden layer with 2^{100} perceptrons in the case of a 100-feature input is near to impossible.
- Perceptron uses a step function as activation function. Sigmoid neuron uses sigmoid function as activation function, which ensures a smoother output function $\hat{y}$ and

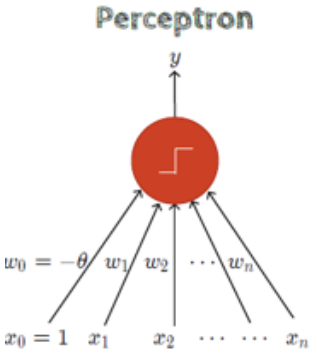
interpretable as a probability value (varies between 0 and 1)



$$\hat{y} = \frac{1}{1 + exp(-z)} \; where \; z = -w_0 + \sum_{i=1}^{n} w_i x_i \tag{5}$$

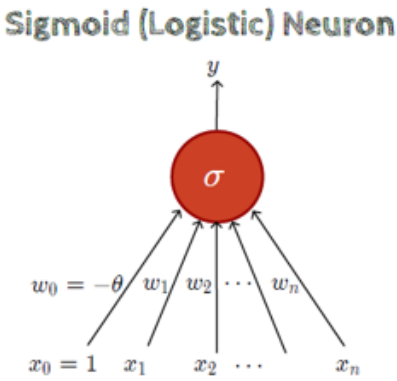
In the above function when z approaches ∞, $\hat{y}$ approaches 1. When z approaches -∞, $\hat{y}$ approaches 0.

- Given below are the pictorial representation of Perceptron and Sigmoid neuron.



$$y = 1 \text{ if } \sum_{i=0}^{n} w_i x_i \geq 0$$

$$= 0 \text{ if } \sum_{i=0}^{n} w_i x_i < 0$$



$$y = \frac{1}{1+exp\left(-\left(w_0 + \sum_{i=1}^{n} w_i x_i\right)\right)}$$

- Since the sigmoid is continuous (and step function is not), it is differentiable and hence can be optimized to geneate the minimum error.

- The goal of machine learning is to find the relation between input x and output y vector. Thus, given input vector x, find the weights w such that $\hat{y}$ simulates a logistic regression (sigmoid function) or linear regression or a quadratic polynomial.
- It's important to note that the weights w and the bias b (w_0) remain unchanged for all input values. These are called parameters of the model. When the values of these parameters are substituted in the respective activation functions of the model, the decision boundary is produced. Thus, in the case of perceptron, the decision boundary is $w_0 + w_1x_1 + w_2x_2 = 0$ and in the case of sigmoid neuron, the decision boundary is

$$\frac{1}{1+exp(-z)} \; where \; z = -w_0 + \sum_{i=1}^{n} w_i x_i \tag{6}$$

- Goal of the learning algorithm is to minimize the error (loss) between the actual output (y) and the expected output (calculated from the decision boundary, and called $\hat{y}$). Ideally, we want the y and $\hat{y}$ to be equal. Or, rather the difference between these two quantities to be minimum. We define $\mathcal{L}(w) = \sum_{i=1}^{n} (\hat{y}_i - y_i)^2$ as the error, in the case of linear regression. This is the objective function that we want to minimize.
- Squared error loss (above) is preferred over absolute error loss ($\mathcal{L}(w) = \sum_{i=1}^{n} |\hat{y} - y_i|$), because the former is not differentiable whereas the latter is not.
- Note that $\mathcal{L}$ is a function of w and b. We will represent the collection of w and b using the vector θ and a change in the vector θ as $\Delta\theta$. Following are the set of equations to denote this.

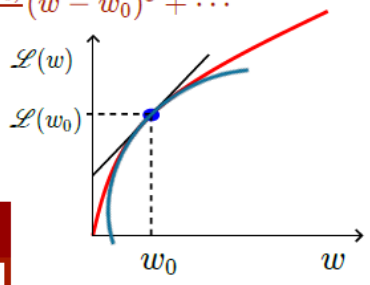
$$\begin{aligned} \theta &= [w, \; b] \\ \Delta\theta &= [\Delta w, \; \Delta b] \\ \theta_{new} &= \theta + \eta\Delta\theta \end{aligned} \tag{7}$$

- Taylor series helps to approximate any function $\mathcal{L}(w)$ within a small neighborhood, if the function's value is known at a certain input w_0, using polynomials of degree n. The higher the degree, better the approximation. For our purposes, we'll use linear approximation.

$$\mathscr{L}(w) = \mathscr{L}(w_0) + \frac{\mathscr{L}'(w_0)}{1!}(w - w_0) + \frac{\mathscr{L}''(w_0)}{2!}(w - w_0)^2 + \frac{\mathscr{L}'''(w_0)}{3!}(w - w_0)^3 + \cdots$$

Linear Approximation ($n = 1$)

$$\mathscr{L}(w) = \mathscr{L}(w_0) + \frac{\mathscr{L}'(w_0)}{1!}(w - w_0)$$



Quadratic Approximation ($n = 2$)

$$\mathscr{L}(w) = \mathscr{L}(w_0) + \frac{\mathscr{L}'(w_0)}{1!}(w - w_0) + \frac{\mathscr{L}''(w_0)}{2!}(w - w_0)^2$$

Note that the approximation will be better if $w - w_0$ is small.

- Using the Taylor's series (linear) approximation, we can write

$$\mathcal{L}(\theta + \eta u) = \mathcal{L}(\theta) + \eta * u^T \nabla_\theta \mathcal{L}(\theta)$$

where $u = \Delta\theta$.

- To make a quadratic approximation, use the formula

$$\mathcal{L}(\theta + \eta u) = \mathcal{L}(\theta) + \eta * u^T \nabla_\theta \mathcal{L}(\theta) + \frac{\eta^2}{2!} * u^T \nabla_\theta^2 \mathcal{L}(\theta) u$$

Note: $\mathcal{L}(\theta)$ is a scalar, $\nabla_\theta \mathcal{L}(\theta)$ is a vector and $\nabla_\theta^2 \mathcal{L}(\theta)$ is a matrix (called hessian).

- This change is considered as favorable only when the loss is reduced from its current value, thus implying $\mathcal{L}(\theta + \eta u) - \mathcal{L}(\theta) < 0$, which in turn implies that $u^T \nabla_\theta \mathcal{L}(\theta) < 0$. The lowest possible value of $u^T \nabla_\theta \mathcal{L}(\theta)$ is when the angle between change in θ is opposite to the current direction of θ.
- Parameter (w *and* b) update rule

$$\begin{aligned} w_{t+1} &= w_t - \eta \nabla w_t \\ b_{t+1} &= b_t - \eta \nabla b_t \\ where\ \nabla w_t &= \frac{\partial \mathcal{L}(w, b)}{\partial w}\ at\ w = w_t,\ b = b_t \\ \nabla b_t &= \frac{\partial \mathcal{L}(w, b)}{\partial b}\ at\ w = w_t,\ b = b_t \end{aligned} \tag{8}$$

∇w_t and ∇b_t is gradient of the loss function evaluated at the current values of w and b

- Above update rule written as pseudo code.

$t \leftarrow 0;$
max_iterarions $\leftarrow 1000;$
w,b $\leftarrow$ initialize randomly
while $t <$ max_iterations do
    $w_{t+1} \leftarrow w_t - \eta \nabla w_t;$
    $b_{t+1} \leftarrow b_t - \eta \nabla b_t;$
    $t \leftarrow t + 1;$
end

Figure 21: Gradient descent algorithm

- Previously, we mentioned squared error loss is used to calculate the loss due to a sigmoid neuron. Assuming that there's only one data point and the activation function is sigmoid, the loss is given by

$$\mathcal{L}(w,\ b)\ =\ \frac{1}{2} * (f(x)\ -\ y)^2$$

$$\nabla w\ =\ \frac{\partial \mathcal{L}(w,b)}{\partial w}\ =\ \frac{\partial}{\partial w}\left[\frac{1}{2} * (f(x)\ -\ y)^2\right]\ =\ (f(x)\ -\ y) * \frac{\partial}{\partial w}(f(x))$$
$$=\ (f(x)\ -\ y) * f(x) * (1\ -\ f(x)) * x$$

$$It\ also\ follows\ that\ \nabla b = (f(x)\ -\ y) * f(x) * (1\ -\ f(x))$$

- If there are more than one data point,

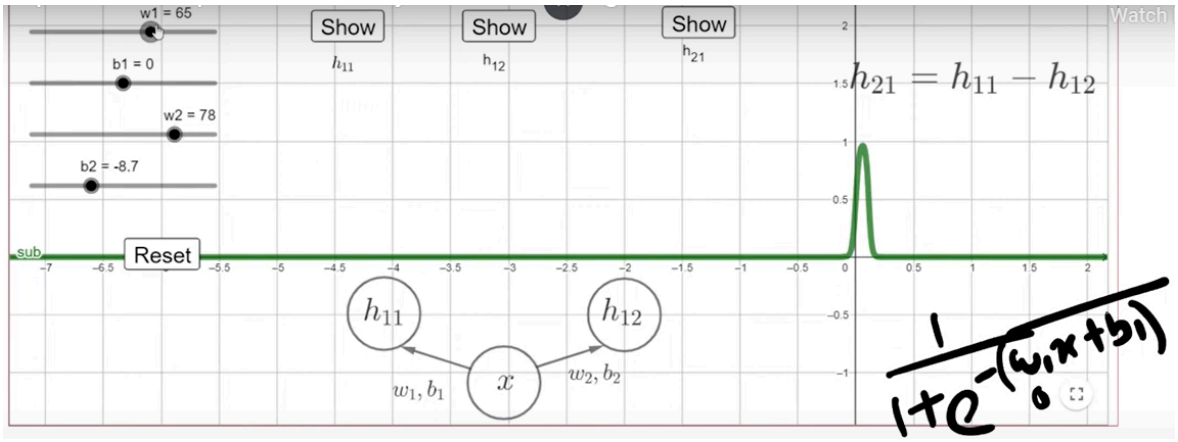
$$\nabla w\ =\ \sum_{i=1}^{N} (f(x_i)\ -\ y_i) * f(x_i) * (1\ -\ f(x_i)) * x_i \tag{9}$$

$$and$$

$$\nabla b\ =\ \sum_{i=1}^{N} (f(x_i)\ -\ y_i) * f(x_i) * (1\ -\ f(x_i))$$

- Note that the gradient ∇w is essentially proportional to the input x. Thus, as x increases, ∇w increases and vice-versa. More importantly, when x is sparse, $\nabla w = 0$.
- A multilayer network of neurons with a single hidden layer can be used to approximate any continuous function (that maps $\mathbb{R}^n \rightarrow \mathbb{R}^m$) to any desired precision.
- For a single input case, this is made possible by constructing tower functions of varying heights and displacement along the X-axis, all of which add up to replicate the original function.
- Note that if we set w to a very high value in a logistic function, we can approximate a step function. We will create two such step functions each using a logistic function

and the second offset from first, using a hidden layer consisting of two neurons. If we subtract these outputs (using a sigmoid neuron with weights 1 and -1), we can simulate a tower function.



- To create a tower with inputs from $\mathbb{R}^2$, we must use 4 sigmoid neurons, each one to create the walls of a 3D tower.
- To create n 2D towers, we need $(2n + 1)$ neurons.
- To create n 3D towers, we need $(5n + 1)$ neurons.

Problems:

Question Number : 47 Question Id : 640653739600 Question Type : SA Calculator : None

Response Time : N.A Think Time : N.A Minimum Instruction Time : 0

Correct Marks : 4

Question Label : Short Answer Question

The logistic sigmoid neuron $\sigma(x)$ is defined as follows

$$\sigma(x) = \frac{1}{1 + exp(-(wx + b))}$$

where $w, b \in \mathbb{R}$ are learnable parameters.Take Mean Square Error loss where required

$$L = 0.5 * (\hat{y} - y)^2$$

Suppose we use the sigmoid function to fit the pair $x = 0, y = 1$, where x is an input and y is the ground truth. Suppose that w is initialized to $w = 2$ and b is initialized to $b = 1$.The prediction $\hat{y}$ by the model for the current w, b is, $\hat{y} = 0.731$. Update the parameter once by keeping $\eta = 10$ and compute the loss. Enter the new loss value.

Note: Enter the loss value to three significant digits. That is, if your answer is 0.06134, then enter it as 0.061

Response Type : Numeric

Evaluation Required For SA : Yes

Show Word Count : Yes

Answers Type : Range

Text Areas : PlainText

Possible Answers :

0.012 to 0.020

- Refer to https://discourse.onlinedegree.iitm.ac.in/t/q47-quiz-1/122392 for solution

  In this problem, $\nabla_w \mathcal{L} = \frac{\partial \mathcal{L}}{\partial w} = \frac{\partial \mathcal{L}}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial w}$ and $\nabla_b \mathcal{L} = \frac{\partial \mathcal{L}}{\partial b} = \frac{\partial \mathcal{L}}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial b}$

  Given that $\hat{y} = \frac{1}{1 + exp(-(-wx + b))}$ and $\mathcal{L} = 0.5(\hat{y} - y)^2$

  From the above loss equation,

$$\frac{\partial \mathcal{L}}{\partial \hat{y}} = 2 * 0.5 * (\hat{y} - y) * \left(\frac{\partial \hat{y}}{\partial \hat{y}} - \frac{\partial y}{\partial \hat{y}}\right) = (\hat{y} - y) * (1 - 0) = (\hat{y} - y)$$

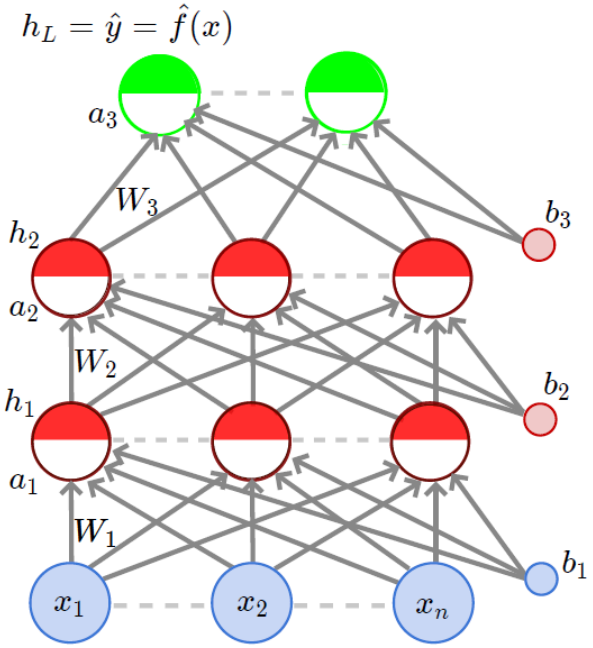
From the equation for $\hat{y}$,

- $\frac{\partial \hat{y}}{\partial w} = \hat{y}(1 - \hat{y}) * \frac{\partial (wx + b)}{\partial w} = \hat{y}(1 - \hat{y}) * x$
- $\frac{\partial \hat{y}}{\partial b} = \hat{y}(1 - \hat{y}) * \frac{\partial (wx + b)}{\partial b} = \hat{y}(1 - \hat{y}) * 1 = \hat{y}(1 - \hat{y})$

Thus, $\frac{\partial \mathcal{L}}{\partial w} = (\hat{y} - y) * \hat{y}(1 - \hat{y}) * x$; $\frac{\partial \mathcal{L}}{\partial b} = (\hat{y} - y) * \hat{y}(1 - \hat{y})$

Week3 - Feed-forward network and backpropagation

- Given below is the schematic for feed-forward network.



The input layer is denoted as h_0, the output layer as h_L and the hidden layers as h_1, h_2. Input layer and each hidden layer has 3 neurons each in this network, though each layer could have different number of neurons - n_0, n_1, n_2. Each layer has been further subdivided into a preactivation layer (represented using $a_1 \ldots a_n$) and activation layer (represented using $h_1 \ldots h_n$).

- In the above network, each layer has a weight vector and a bias vector associated with it. Since the first layer is connected to 3 nuerons in the previous layer, you have 3

x 3 connections, and hence the weight vector W_1 is a 3 x 3 matrix. b_1 is denoted as a vector in $\mathbb{R}^3$. Similarly, W_2 is a 3 x 3 matrix. b_2 is denoted as a vector in $\mathbb{R}^3$. The output layer only has 2 neurons, each of which is connected to 3 neurons in the previous layer. Hence, W_3 is a 2 x 3 matrix. b_1 is denoted as a vector in $\mathbb{R}^2$.

- The preactivation vector at the i^{th} layer is given by

$$a_i(x) = b_i + W_i h_{i-1}(x)$$

In terms of vectors and matrices, the above equation can be rewritten for the first layer as

$$a_1 = \begin{bmatrix} a_{11} \\ a_{12} \\ a_{13} \end{bmatrix} = \begin{bmatrix} b_{11} \\ b_{12} \\ b_{13} \end{bmatrix} + \begin{bmatrix} W_{111} & W_{112} & W_{113} \\ W_{121} & W_{122} & W_{123} \\ W_{131} & W_{132} & W_{133} \end{bmatrix} \begin{bmatrix} h_{01} \\ h_{02} \\ h_{03} \end{bmatrix} \tag{10}$$

Note that $\begin{bmatrix} h_{01} \\ h_{02} \\ h_{03} \end{bmatrix}$ is another way of representing input vector $\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$.

- Activation vector at the i^{th} layer is given by

$$h_i(x) = g(a_i(x))$$

In terms of vectors and matrices, and assuming that g is a sigmoid function, the above equation can be rewritten for the first layer as

$$h_1 = \begin{bmatrix} h_{11} \\ h_{12} \\ h_{13} \end{bmatrix} = g\left(\begin{bmatrix} a_{11} \\ a_{12} \\ a_{13} \end{bmatrix}\right) = \begin{bmatrix} g(a_{11}) \\ g(a_{12}) \\ g(a_{13}) \end{bmatrix} = \begin{bmatrix} \frac{1}{1+e^{-a_{11}}} \\ \frac{1}{1+e^{-a_{12}}} \\ \frac{1}{1+e^{-a_{13}}} \end{bmatrix} \tag{11}$$

Due to eq.(10), we can substitute $a_{11} = b_{11} + W_{111}*h_{01} + W_{112}*h_{02} + W_{113}*h_{03}$ in eq. (11). Similarly for a_{12} *and* a_{13}.

- At the output layer,

$$f(x) = h_3 = \begin{bmatrix} h_{31} \\ h_{32} \\ h_{33} \end{bmatrix} = O\left(\begin{bmatrix} a_{31} \\ a_{32} \\ a_{33} \end{bmatrix}\right)$$

Note that the function O can't be a sigmoid function, since sigmoids are generally

used for binary classification. It also can't be a linear function, since output of a linear function isn't bounded. Typically, O is a softmax function in the case of multi-class classification problems represented as follows.

$$\hat{y}_j = O(a_L)_j = \frac{e^{a_{L,j}}}{\sum_{i=1}^{k} e^{a_{L,i}}}$$

Note here that k is the number of neurons in the output layer.

- The entire network can be represented mathematically as

Data: $\{x_i, y_i\}_{i=1}^{N}$

Model:

$$\hat{y}_i = \hat{f}(x_i) = O(W_3 g(W_2 g(W_1 x + b_1) + b_2) + b_3)$$

Parameters:

$$\theta = W_1, ..., W_L, b_1, b_2, ..., b_L (L = 3)$$

- Will use gradient descent with backpropagation to learn the parameters, so as to minimize the (square-error) loss.

Objective/Loss/Error function: Say,

$$min \frac{1}{N} \sum_{i=1}^{N} \sum_{j=1}^{k} (\hat{y}_{ij} - y_{ij})^2$$

$$In\ general, min\ \mathscr{L}(\theta)$$

where $\mathscr{L}(\theta)$ is some function of the parameters

Note here that k is the number of neurons in the output layer, and N is the number of data points.

- Gradient descent when appied to this network will be much more complex, since it involves taking the gradient of the loss with respect to n^2 weights in addition to the n biases in each layer.
- Choice of the loss function depends on the nature of the problem. For regression problems, squared-error loss is typical. However, for classification problems, cross-entropy loss is preferred. Cross-entropy loss is given by the following formula

$$\mathscr{L}(\theta) = -\sum_{c=1}^{k} y_c \log \hat{y}_c$$

Note that in the above formula, $y_c = 1$ for the correct class (say l) and 0 for all other

classes. Hence this evaluates to $\mathcal{L}(\theta) = -log\left(\widehat{y_l}\right)$, where l is the true class label. This is often referred to as the *negative log likelihood*.

- We must minimize the *negative log likelihood* during the gradient descent process. This means, we must maximize the *log likelihood*, $log\left(\widehat{y_l}\right)$.
- Here is a table that contains the preferrred output activation and loss function, for a regression as well as a classification problem.

	Outputs	
	Real Values	Probabilities
Output Activation	Linear	Softmax
Loss Function	Squared Error	Cross Entropy

- To understand how backpropagation works, consider this network.

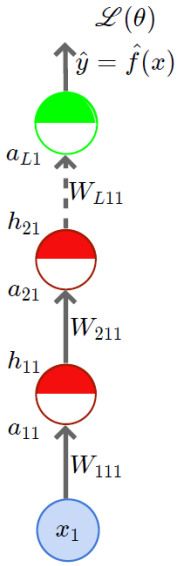


Figure 22: A very thin network

From the above network, we can use the chain rule to find the derivative of the loss $\mathcal{L}$ with respect to W_{111} as

$$\frac{\partial \mathscr{L}(\theta)}{\partial W_{111}} = \frac{\partial \mathscr{L}(\theta)}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial a_{L11}} \frac{\partial a_{L11}}{\partial h_{21}} \frac{\partial h_{21}}{\partial a_{21}} \frac{\partial a_{21}}{\partial h_{11}} \frac{\partial h_{11}}{\partial a_{11}} \frac{\partial a_{11}}{\partial W_{111}}$$

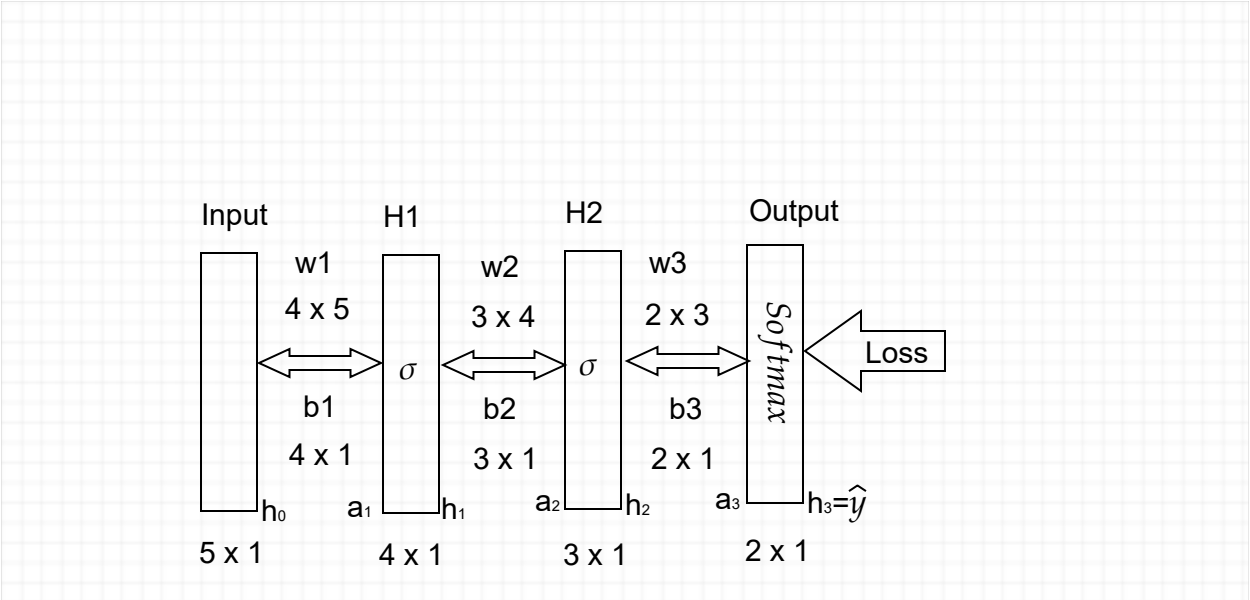
$$\frac{\partial \mathscr{L}(\theta)}{\partial W_{111}} = \frac{\partial \mathscr{L}(\theta)}{\partial h_{11}} \frac{\partial h_{11}}{\partial W_{111}} \quad \text{(just compressing the chain rule)}$$

Note that we will be able to reuse parts that have already been computed.

- Another way to get the inutition behind backpropagation is to consider the chaining of responsibilities in the backward direction, as represented in the figure below.

$$\underbrace{\frac{\partial \mathcal{L}(\theta)}{\partial W_{111}}}_{\text{Talk to the weight directly}} = \underbrace{\frac{\partial \mathcal{L}(\theta)}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial a_3}}_{\text{Talk to the output layer}} \underbrace{\frac{\partial a_3}{\partial h_2} \frac{\partial h_2}{\partial a_2}}_{\text{Talk to the previous hidden layer}} \underbrace{\frac{\partial a_2}{\partial h_1} \frac{\partial h_1}{\partial a_1}}_{\text{Talk to the previous hidden layer}} \underbrace{\frac{\partial a_1}{\partial W_{111}}}_{\text{and now talk to the weights}}$$

- In the case of a deep neural network with 2-hidden layers, remember these formulae



- Forward propagation:
  - $h_0 = \sigma(a_0)$. a_0 is same as input vector.
  - $a_1 = w_1 @ h_0 + b_1$
  - $h_1 = \sigma(a_1) = \dfrac{1}{1 + e^{-a_{1i}}}$
  - $a_2 = w_2 @ h_1 + b_2$
  - $h_2 = \sigma(a_2) = \dfrac{1}{1 + e^{-a_1}}$
  - $a_3 = w_3 @ h_2 + b_3$
  - $h_3 = Softmax(a_3) = \dfrac{e^{a_{3i}}}{e^{a_{31}} + e^{a_{32}}}$
- Backward propagation:

$$\sigma(z) = \frac{1}{1+e^{-z}}$$
$$\sigma'(z) = \sigma(z)(1-\sigma(z))$$

$$\nabla_{\hat{y}}(L) = -\frac{e_l}{\hat{y}_l} \in \mathbb{R}^k$$
$$\nabla_{a_L}(L) = -(e_l - \hat{y}) \in \mathbb{R}^k$$
$$\nabla_{h_i}(L) = W_{i+1}^T(\nabla_{a_{i+1}} L) \in \mathbb{R}^m$$
$$\nabla_{a_i}(L) = \nabla_{h_i}(L) \odot \sigma'(a_i) \in \mathbb{R}^m$$
$$\nabla_{W_i}(L) = (\nabla_{a_i} L).h_{i-1}^T \in \mathbb{R}^{m \times p}$$

- Alternative formuale (useful in solving problems):

$$\frac{\partial L}{\partial \mathbf{W_{123}}} = \frac{\partial L}{\partial \mathbf{a_1}} . \frac{\partial \mathbf{a_1}}{\partial \mathbf{W_{123}}}$$

$$\frac{\partial L}{\partial \mathbf{a_1}} = \frac{\partial L}{\partial \mathbf{h_1}} \odot \sigma'(a_1)$$

$$\frac{\partial L}{\partial \mathbf{h_1}} = \mathbf{W_2}^T . \frac{\partial L}{\partial \mathbf{a_2}}$$
$$\frac{\partial L}{\partial \mathbf{a_2}} = \frac{\partial L}{\partial \mathbf{h_2}} \odot \sigma'(a_2)$$

$$\frac{\partial L}{\partial \mathbf{h_2}} = \mathbf{W_3}^T . \frac{\partial L}{\partial \mathbf{a_3}}$$

- Older formulae (from Lalit)
  - $\nabla_{a_3}^{\mathcal{L}} = \hat{y} - y$
  - $\nabla_{w_3}^{\mathcal{L}} = (\hat{y} - y) \mathbin{@} h_2^T$
  - $\nabla_{b_3}^{\mathcal{L}} = \hat{y} - y$
  - $\nabla_{w_2}^{\mathcal{L}} = \left[w_3^T \mathbin{@} (\hat{y} - y) \odot h_2(1 - h_2) \right] \mathbin{@} h_1^T.$
  - $\nabla_{b_2}^{\mathcal{L}} = w_3^T \mathbin{@} (\hat{y} - y) \odot h_2(1 - h_2)$

    Now, let $A = \left[w_3^T \mathbin{@} (\hat{y} - y) \odot h_2(1 - h_2) \right]$, first part in the above equation.
  - $\nabla_{w_1}^{\mathcal{L}} = \left[w_2^T \mathbin{@} A \odot h_1(1 - h_1) \right] \mathbin{@} h_0^T$

- $\nabla_{b_1}^{\mathfrak{L}} = w_2^T @ A \odot h_1(1 - h_1)$

Note the differences in these computations in comparison to that given in detailed notes. As an example, $\nabla_{w_3}^{\mathfrak{L}} = (\hat{y} - y) @ {h_2}^T$ in this document, whereas in the notes, $\nabla_{w_3}^{\mathfrak{L}} = h_2 @ (\hat{y} - y)^T$. This is because the weight matrices are transposes of each other. Thus, w3 is assumed to be 2 x 3 in this document, whereas in the detailed notes, it's assumed to be 3 x 2. This is a very important distinction.

- Weight/bias updates:
  - $w_3 = w_3 - \eta\nabla_{w_3}^{\mathfrak{L}}$
  - $b_3 = b_3 - \eta\nabla_{b_3}^{\mathfrak{L}}$
  - $w_2 = w_2 - \eta\nabla_{w_2}^{\mathfrak{L}}$
  - $b_2 = b_2 - \eta\nabla_{b_2}^{\mathfrak{L}}$
  - $w_1 = w_1 - \eta\nabla_{w_1}^{\mathfrak{L}}$
  - $b_1 = b_1 - \eta\nabla_{b_1}^{\mathfrak{L}}$

Week4 - Improved gradient descent algorithms

- Contours are 2D representations of geometric shapes in 3D.
- The 3D shapes are sliced horizontally at equal intervals along z-axis to obtain contours. Each contour ring is typically marked with the loss value.
- When the 3D shapes have higher slope, the contour rings are closeby. When the 3D shapes have gentle slope, the contour rings are far apart.
- Momentum-based GD: If the direction of movement is same during consecutive updates, move faster.
- In MGD, past updates are accumulated as $u_t = \beta u_{t-1} + \nabla w_t$. It's assumed that $0 \leqslant \beta \leqslant 1$, $u_{-1} = 0$ and w_0 is a random vector/matrix. This leads to the following calculations.

$$u_0 = (\beta * u_{-1}) + \nabla w_0 = (\beta * 0 + \nabla w_0) = \nabla w_0$$
$$u_1 = \beta u_0 + \nabla w_1 = \beta * \nabla w_0 + \nabla w_1$$
$$u_2 = \beta u_1 + \nabla w_2 = \beta(\beta * \nabla w_0 + \nabla w_1) + \nabla w_2 = \beta^2 \nabla w_0 + \beta \nabla w_1 + \nabla w_2 \quad (12)$$

In general, $u_t = \sum_{i=0}^{t} \beta^{t-i} \nabla w_i$

- The update rule for MGD is as follows:

$$u_t = \beta u_{t-1} + \nabla w_t$$
$$w_{t+1} = w_t - \eta u_t \quad (13)$$

Note that this pair of equations is used instead of the equation for normal gradient descent, $w_{t+1} = w_t - \eta \nabla w_t$

- Substituting the value of ∇u_2 from eq.(12) in eq.(13), we get,

$$w_1 = w_0 - \eta u_0 = w_0 - \eta(\nabla w_0)$$
$$w_2 = w_1 - \eta u_1 = w_1 - \eta(\beta \nabla w_0 + \nabla w_1)$$
$$w_3 = w_2 - \eta u_2 = w_2 - \eta\left(\beta^2 \nabla w_0 + \beta \nabla w_1 + \nabla w_2\right)$$

- MGD could cause faster movements, but it could be faster than desired and hence might miss target. THis could happen multiple times, each time requiring to take u-turns, so as to ultimately reach the target.
- Some observations:
  - Setting $\beta = 0.1$ allows the algorithm to move faster than vanilla (plain) gradient descent algorithm
  - Setting $\beta = 0$ makes it equivalent to vanilla gradient descent algorithm
  - Oscillation around the minimum will be less if we set $\beta = 0.1$ than setting $\beta = 0.99$
- Nesterov's Gradient Descent (NAG) is used to counter this, the idea being that during each update, look ahead and decide what should the final update be. Thus, instead of $u_t = \beta u_{t-1} + \nabla w_t$, the modified update rule is $u_t = \beta u_{t-1} + \nabla(w_t - \beta u_{t-1})$. This helps NAG in correcting its course quicker than MGD. The second term is called the look-ahead value.
- The update rule for NAG is as follows:

$$u_t = \beta u_{t-1} + \nabla(w_t - \beta u_{t-1})$$
$$w_{t+1} = w_t - \eta u_t \tag{14}$$

- In the gradient descent algorithm (code below), gradients are calculated for each data point and accumulated before the weight update is performed once. This is repeated for max_epoch iterations.

```
import numpy as np
X = [0.5,2.5]
Y = [0.2,0.9]



def do_gradient_descent():

  w,b,eta,max_epochs = -2,-2,1.0,1000

  for i in range(max_epochs):
    dw,db = 0,0
    for x,y in zip(X,Y):
      dw += grad_w(x,w,b,y)
      db += grad_b(x,w,b,y)

    w = w - eta*dw
    b = b - eta*db
```

The above algorithm calculates true gradients. It performs one weight update per epoch. This is ideal, but when the number of data points are large (say, a million), then this would be prohibitively expensive to calculate the gradient for a million datapoints and also convergence would take extremely long time.

- To counter this, two other algorithms are used, both of which computes approximate gradients instead of true gradients - stochastic gradient descent (weight update after every data point) and mini-batch gradient descent (weight update after a $batch$ of data).
- SGD performs N weight updates per epoch, MBGD performs N/B weight updates per epoch, where N is the number of data points, and B is the mini-batch size.
- SGD computes gradient at every data point, causing a lot of oscillations/instability at every weight update. MBGD also causes oscillations in its weights updates, but not as much as SGD.
- In practice, MBGD is the most commonly used algorithm.
- It's logical to increase the learning rate to achieve the convergence faster, because this could result in oscillations and hence not practical.
- There are various strategies to anneal learning rates
  - Consistently reduce the learning rate after a fixed number of epochs
  - Rerun the epoch with a reduced learning rate, until validation error reduces from one epoch to the next.
  - Exponential decay of learning rate
    - $\eta = \eta_0^{-kt}$.
    - $\eta = \frac{\eta_0}{1+kt}$
    - Note: η_0 and k are hyper-parameters, and t is the step number.
  - Linear search. Try out each from among a set of learning rates, choose the learning rate that gives the least loss. Note that the weight update is performed only once.

- For convex loss functions, vanilla (batch) gradient descent is guaranteed to eventually converge to the global optimum, while stochastic gradient descent is not.

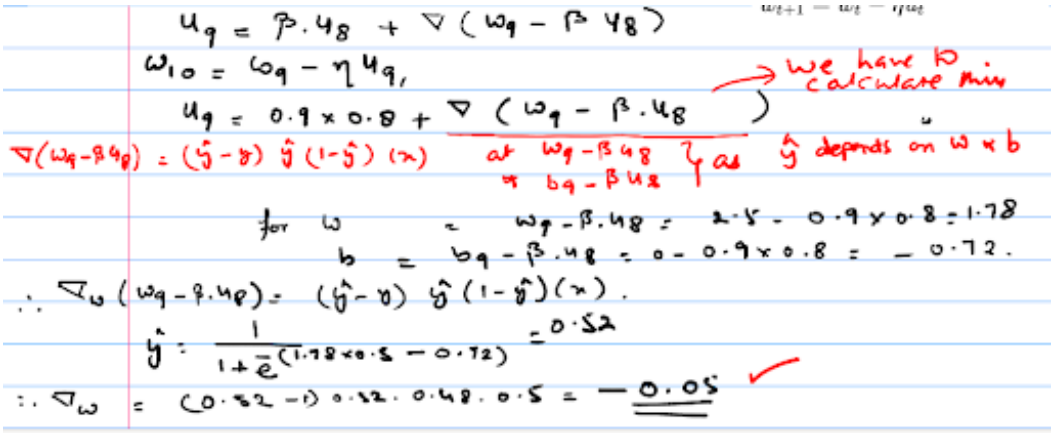
Problems

5) Consider a Sigmoid neuron with a single input. The value of input $x = 0.5$ and the true output value $y = 1$. The weight and bias of the neuron are initialized to $w_0 = 5$ and $b_0 = -5$. The model uses Nesterov Accelerated gradient descent algorithm with $\beta = 0.9$ and $\eta = 0.1$. Also,it uses Mean Square Error (MSE) loss of the form $L(w) = \frac{1}{2}(\hat{y} - y)^2$.

Suppose that the model has been trained for 10 iterations (iteration starts from zero). The state of the parameters at $10^{th}(t = 9)$ iteration is as follows, $u_8 = 0.8, w_9 = 2.5, b_9 = 0$

- Calculate the gradient of look-ahead value for the next weight update.

Following is the solution:



For details, refer to https://discourse.onlinedegree.iitm.ac.in/t/week-4-ga-q5/119804

Week5 - Adaptive learning rate

- Assuming that we've a multi-dimensional input vector, say $\{x_1,\ x_2,\ x_3,\ x_4\}$, we can represent the associating gradient vector as $\{\nabla w_1,\ \nabla w_2,\ \nabla w_3,\ \nabla w_4\}$, where

$$\nabla w_1 = (f(x) - y) * f(x)(1 - f(x)) * x_1$$
$$\ldots$$
$$\nabla w_4 = (f(x) - y) * f(x)(1 - f(x)) * x_4$$

- If there are n datapoints, in order to find the gradient of a specific feature, say the second feature x_2 we can just sum the gradients over all n points to get the total gradient like $\sum_{i=1}^{m} (f(x) - y) * f(x)(1 - f(x)) * x_i^2$
- If a feature is sparse, the loss derivative and hence the weight update will be small for the feature. Thus, the weight updates for this feature will be relatively small and more pronounced for the denser features. One way to get over this anomaly is to have a larger learning rate for sparse features, in comparison to the denser features.
- The update rule for AdaGrad is as follows:

$$v_t = v_{t-1} + (\nabla w_t)^2$$

$$w_{t+1} = w_t - \frac{\eta_0}{\sqrt{v_t + \epsilon}} * \nabla w_t \tag{15}$$

Note that the sparser the feature is, v is building up slowly, and hence the learning rate η will decay slowly due to the self-correction mechanism $\frac{\eta_0}{\sqrt{v_t + \epsilon}}$, causing slower weight updates. On the other hand, the denser the feature is, v builds up faster, and hence the effective learning rate will be decay rapidly due to the self-correction mechanism mentioned above, causing rapid weight updates.

- The below graph is plotted for sparse features. Note that ∇w is increases slightly on the negative and flattens out soon at around 100 iterations, v_t keeps increasing, since it's an accumulation of square of gradients, and flattens at around 80 iterations. The effective learning rate (shown on the right side) decays slowly through iterations.

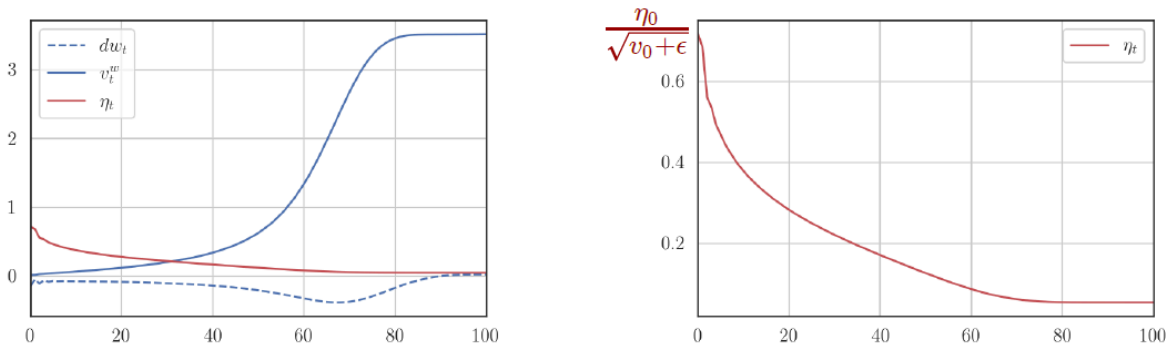


Figure 23: AdaGrad - Sparse feature

- The below graph is plotted for dense features. Note that the scae of the graph is much higher due to the rapid growth of v_t and hence the ∇w plot is invisible. The effective learning rate (shown on the right side) decays rapidly through iterations.

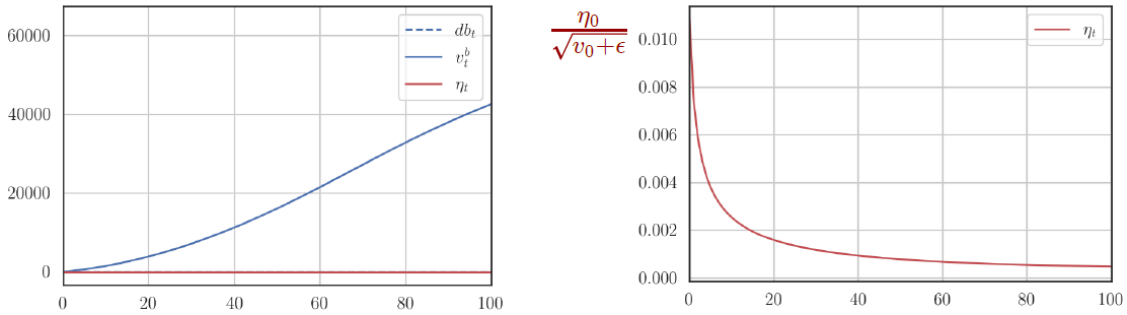


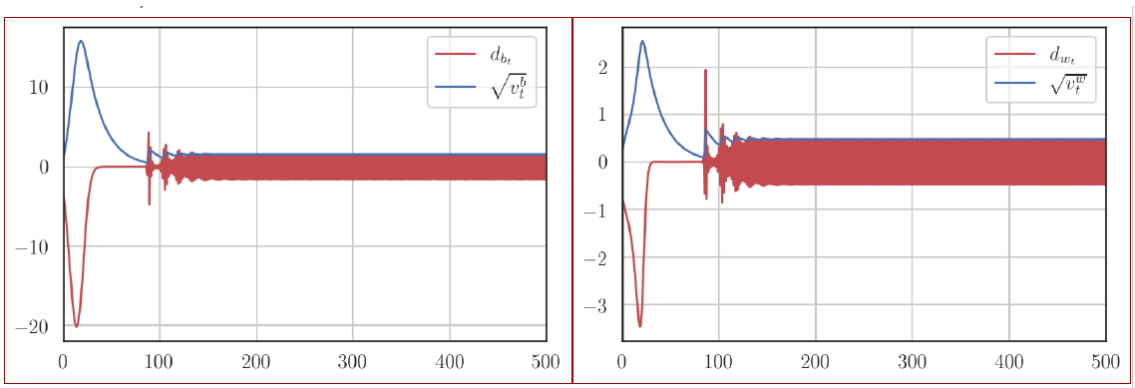
Figure 24: AdaGrad - Dense feature

- But, there's one downside of Adagrad, especially with dense features. As can be observed in the graph, when it reaches the minima of the loss function, the gradient is close to 0, but the effective learning rate is also very small (due to high velocity) which causes the updates to happen extremely slowly. This is in contrast with the reason Adagrad was invented in the first place.
- To avoid this issue of fast diminishing learning rates, we can reduce the velocity - both current and past, by a factor of $1-\beta$ and β respectively.
- Hence, the update rule for RMSprop is as follows:

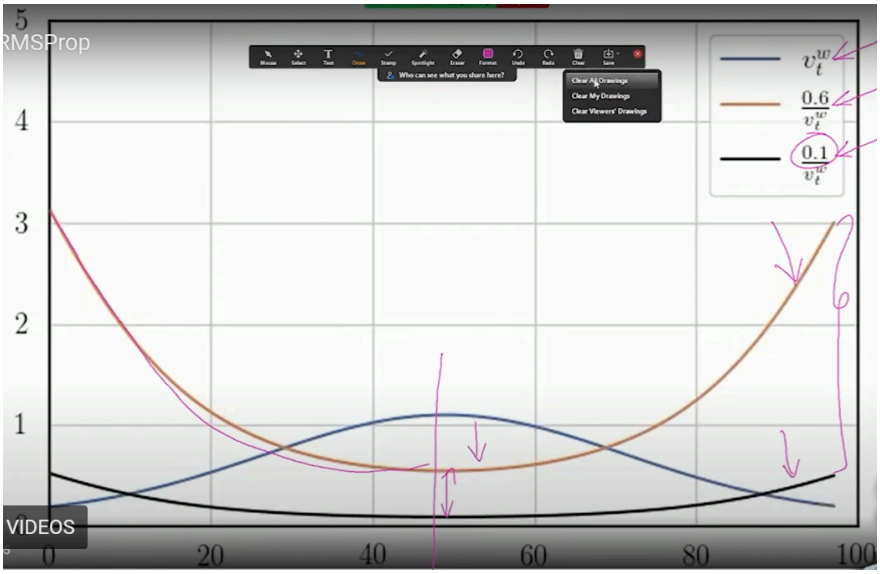
$$v_t = \beta v_{t-1} + (1-\beta)(\nabla w_t)^2$$
$$w_{t+1} = w_t - \frac{\eta_0}{\sqrt{v_t+\epsilon}} * \nabla w_t \quad (16)$$

The above rule ensures that, for dense features, learning rate decays slowly, instead of rapidly (as in AdaGrad)

- On the downside, RMSprop causes the learning rate tends to become a *constant* after a few iterations. This happens because the velocity, unlike AdaGrad, increases in regions where the gradient is increasing, but starts reducing when the gradient starts to decrease before flattening entirely. This is because of the β factor that gets progressively multiplied with the previous velocities - $\beta,\ \beta^2,\ \beta^3 \ldots \beta^k$. Recall, in the case of AdaGrad, the velocity always keeps increasing, until it reaches the minima of the loss function.



- One solution is to set the initial learning rate carefully, so that the *symmetric* oscillations around the minima wouldn't happen. But, this isn't always practical. TYpically, smaller initial learning rates helps reduce the oscillations around the minima.



- Thus, RMSprop can be said to be sensitive to the initial learning rate, without which oscillations would occur towards the minima of the loss function. However, this behavior is not always desirable. Thus, in steep regions, it's better to have low initial learning rate, but in flat regions, it's better to have a high initial learning rate. See below.

Which one is better?

in steep regions, say, $v_t = 1.25$	in flat regions, say, $v_t = 0.1$
$\eta_t = \frac{0.6}{1.25} = 0.48$	$\eta_t = \frac{0.6}{0.1} = 6$
$\eta_t = \frac{0.1}{1.25} = 0.08$	$\eta_t = \frac{0.1}{0.1} = 1$

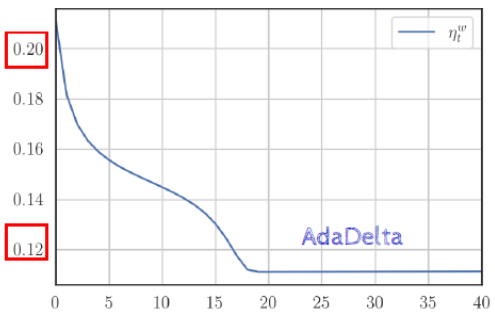
$\eta_t = 0.08$ is better in a steep region and $\eta_t = 6$ is better in a gentle region. Therefore, we wish the numerator also to change wrt gradient/slope

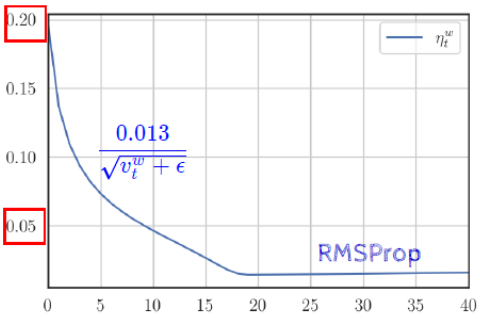
- In AdaDelta, the numerator η_0 in the learning rate computation is replaced a history of $-\sqrt{u_{t-1} + \epsilon}$, where $u_t = \beta u_{t-1} + (1 - \beta)\left(\Delta w_t^2\right)$
- The update rule for AdaDelta is as follows:

$$
\begin{aligned}
v_t &= \beta v_{t-1} + (1-\beta)(\nabla w_t)^2 \\
\Delta w_t &= -\frac{\sqrt{u_{t-1} + \epsilon}}{\sqrt{v_t + \epsilon}} \nabla w_t \\
w_{t+1} &= w_t + \Delta w_t \\
u_t &= \beta u_{t-1} + (1-\beta)(\Delta w_t)^2
\end{aligned}
\tag{17}
$$

Note that u_t computed in the current iteration will be used in the next iteration as u_{t-1}.

- In steep regions, the numerator of Δw_t is smaller than the denominator, which implies that the effective learning rate will be small. In the flat regions, the current velocity v_t will be smaller than the numerator, which implies that the effective learning rate will be high.
- Shown below is the plot of effective learning rate for AdaDelta and RMSprop.





Notice that the learning rate decays rapidly in the case of RMSprop, whereas AdaDelta doesn't and converges a lot faster.

- The update rule for Adam is as follows:

$$
\begin{aligned}
m_t &= \beta_1 m_{t-1} + (1-\beta_1)\nabla w_t \\
\widehat{m_t} &= \frac{m_t}{1-\beta_1^t} \\
v_t &= \beta_2 v_{t-1} + (1-\beta_2)(\nabla w_t)^2 \\
\widehat{v_t} &= \frac{v_t}{1-\beta_2^t} \\
w_{t+1} &= w_t - \frac{\eta_0}{\sqrt{\widehat{v_t} + \epsilon}} * \widehat{m_t}
\end{aligned}
\tag{18}
$$

Note: $\widehat{m_t}$ and $\widehat{v_t}$ are called bias-corrected terms of m_t and v_t respectively. Bias corrections on these terms significantly reduces them.

- $\widehat{v_t}$ used in the above set of rules is an exponentially weighted L2 norm, since it's equivalent to $a\nabla w_0^2 + b\nabla w_1^2$+...$n\nabla w^2$
- Typical values of $\beta_1 = 0.9$ and $\beta_2 = 0.999$
- In the update rule for Adam as used in eq.(18), we're using L_2 norm for computing the velocity vector v_t
- L_p norm of the vector $x = \left(|x_1^p| + |x_2^p| + |x_3^p| ... + |x_n^p|\right)^{\frac{1}{p}}$. Thus, any point that lies on the following shape has its $L_1 = 1$.

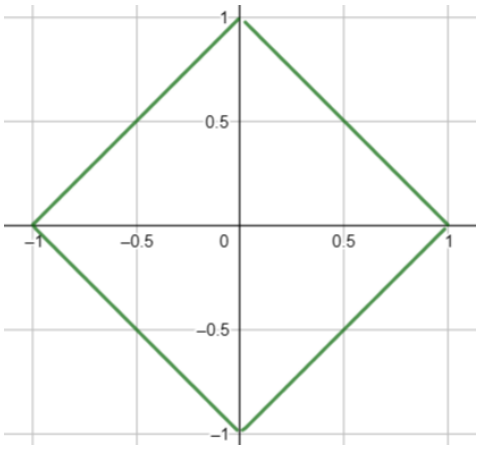


Figure 25: L_1 norm for all points on this unit square = 1

Similarly, any points on a unit circle has its $L_2 = 1$.

- L_2 is the most preferred norm, since higher degrees becomes unstable - higher power

of smaller values of x is extremely small, and the same for larger values of x are extremely large.

- When L_2 is used in the update rule, the learning rate increases whenever zero inputs are encountered, due to the exponentially decaying velocity.

$$v_t = \beta_2 v_{t-1} + (1-\beta_2)(\nabla w_t)^2, \quad \beta_2 = 0.999$$
$$v_0 = 0.999 * 0 + 0.001(\nabla w_0)^2 = 0.001$$
$$\hat{v_0} = \frac{0.001}{1-0.999} = 1$$
$$\eta_t = \frac{1}{\sqrt{1}} = 1$$
$$v_1 = 0.999 * (0.001) + 0.001(0)^2 = 0.000999$$
$$\hat{v_1} = \frac{0.000999}{1-0.999^2} = 0.499$$
$$\eta_t = \frac{1}{\sqrt{0.499}} = 1.41$$

- When $p = \infty$, $L_p = max(x_1, x_2 ... x_n)$. Using the max norm (instead of in the update rule for Adam in eq.(18), we have the following equations for velocity vector and the weight update.

$$v_t = \max(\beta_2^{t-1}|\nabla w_1|, \beta_2^{t-2}|\nabla w_2|, \cdots, |\nabla w_t|)$$
$$v_t = \max(\beta_2 v_{t-1}, |\nabla w_t|)$$
$$w_{t+1} = w_t - \frac{\eta_0}{v_t}\nabla w_t$$

Note that when we use max norm, we don't need to do bias correction. This algorithm is called *AdaMax*. When max norm is applied to *RMSprop*, the resulting algorithm is called *MaxProp*.

- The above strategy ensures that the learning rate doesn't get modified when encountering zero inputs.

$$v_t = \max(\beta_2 v_{t-1}, |\nabla w_t|), \quad \beta_2 = 0.999$$
$$v_0 = \max(0, |\nabla w_0|) = 1$$
$$\eta_t = \frac{1}{1} = 1$$
$$v_1 = \max(0.999 * 1, 0|) = 0.999$$
$$\eta_t = \frac{1}{0.999} = 1.001$$
$$v_2 = \max(0.999, 1) = 1$$
$$\eta_t = \frac{1}{1} = 1$$

- The update rule for AdaMax is as follows:

$$m_t = \beta_1 m_{t-1} + (1-\beta_1)\nabla w_t$$

$$\widehat{m_t} = \frac{m_t}{1-\beta_1^t}$$
$$v_t = max(\beta_2 v_{t-1}, \nabla w_t)$$
$$w_{t+1} = w_t - \frac{\eta_0}{v_t + \epsilon} * \widehat{m_t} \tag{19}$$

- Hence, the update rule for MaxProp is as follows:

$$v_t = max(\beta v_{t-1}, \nabla w_t)$$
$$w_{t+1} = w_t - \frac{\eta_0}{v_t + \epsilon} * \nabla w_t \tag{20}$$

- When the Nesterov effect is added to Adam, we get NAdam.
- The update rule for NAdam is as follows:

$$m_t = \beta_1 m_{t-1} + (1-\beta_1)\nabla w_t$$
$$\widehat{m_t} = \frac{m_t}{1-\beta_1^t}$$
$$v_t = \beta_2 v_{t-1} + (1-\beta_2)(\nabla w_t)^2$$
$$\widehat{v_t} = \frac{v_t}{1-\beta_2^t}$$
$$w_{t+1} = w_t - \frac{\eta_0}{\sqrt{\widehat{v_t} + \epsilon}} * \left(\beta_1 \widehat{m_{t+1}} + \frac{(1-\beta_1)\nabla w_t}{1-\beta_1^{t+1}}\right) \tag{21}$$

- Learning rate schemes could be based on epochs, validation, or gradients.
- Epoch based schemes - Step decay, Exponential decay, Cyclical, Cosine annealing, Warm restart
- Validation based schemes - Line search, Log search
- Gradient based schemes - AdaGrad, RMSprop, AdaDelta, Adam, AdaMax, NAdam, AMSGrad, AdamW

Week6 - Bias, Variance, Regularization

- Deep learning models typically have more parameters than there are samples and hence prone to overfitting, which means that the model could memorize all training samples, but poorly generalize with respect to test data.
- Assume that we're trying to fit a set of 500 samples to a simple model $y = \widehat{f(x)} = w_1 x + w_0$ and to a complex model $y = \widehat{f(x)} = \sum_{i=1}^{25} w_1 x^i + w_0$. The normal strategy is to randomly pick m samples ($m < 500$) and use for training these

models. This process will be repeated k times.

- During each iteration, objective is to find a set of weights such that the loss $\sum_{i=1}^{m}\left(y_i - \widehat{f(x_i)}\right)^2$ is minimum. Note that during each iteration, the weights thus obtained will be slightly different as shown below

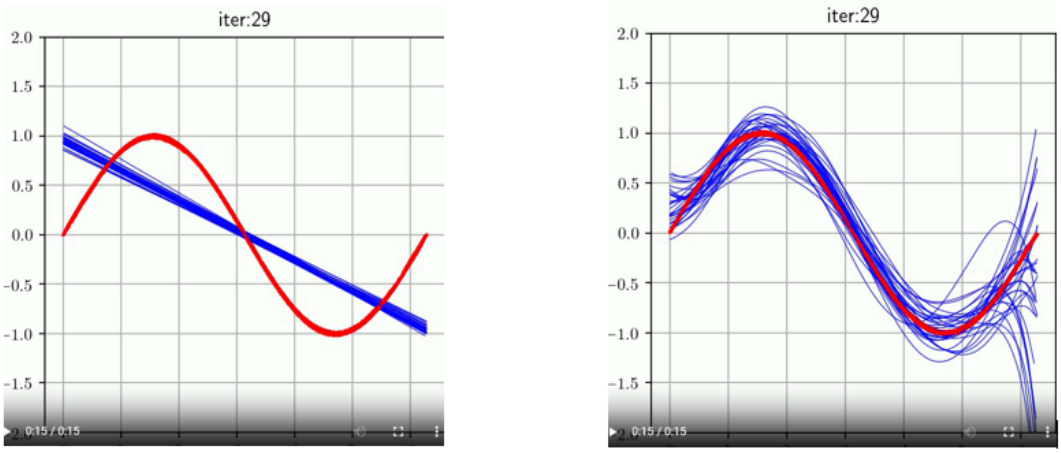


Figure 26: Fitting the samples to a simple model (left) and complex model (right)

- .In the simple model, the fitted lines are very close to each other, but it differs significantly from the true value (underfits). In the complex model, the fitted curves are closer to the true value, but far off from each other.
- $Bias$ is defined as the difference between the average (expectation) from k models and the true value for a given x, $E\left[\widehat{f(x)}\right] - f(x)]$. Simple models have high bias and complex models have low bias.
- $Variance$ is defined as the difference between the average(expectation) from k models and each of the k models, $E\left[\left(\widehat{f(x)} - E\left[\widehat{f(x)}\right]\right)^2\right]$. Simple models have low variance and complex models have high variance.
- Typically, when the model have a low bias, it has a high variance. This is not desirable. Model must have a balance between bias and variance.
- $E\left[\left(f(x) - \widehat{f(x)}\right)^2\right]$ denotes the error between the true value and prediction, for a given x and is related to the bias and variance by the following relationship. $Error = Bias^2 + Variance + \sigma^2$ (The last quantity if called the irreducible error).
- In a supervised ML problem, two types of error exists - train error and test error. Train error, typically reduces as the model complexity increases, since bias is low and variance can be kept in control for known samples. But, test error increases after certain point, because keeping variance in control isn't possible for unseen data.

Thus, as per the above formula, test error increases. Relationship between these entities is as shown below (blue curve represents train error and red curve represents test error)

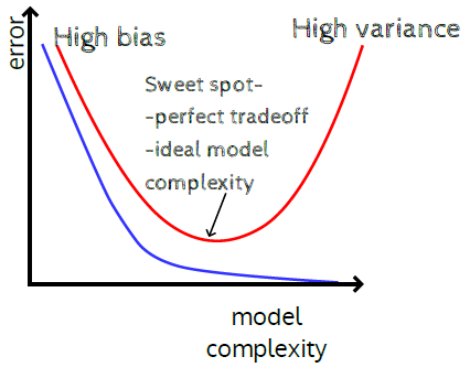


Figure 27: Train and test errors.

- It can be proved mathematically that
  $True\ error\ =\ empiral\ test\ error\ +\ small\ constant$. However,
  $True\ error\ =\ empiral\ train\ error\ +\ small\ constant + E\left[\epsilon\widehat{f(x)} - f(x)\right]$, where ϵ is $y - f(x)$.
- Note that the last term in the above equation can be rewritten as
  $\frac{1}{n}\sum_{i=1}^{n}\epsilon_i\left(\widehat{f(x_i)} - f(x_i)\right)$, which by Stein's lemma equals $\frac{\sigma^2}{n}\sum_{i=1}^{n}\frac{\partial\widehat{f(x_i)}}{\partial y_i}$. This quantity is high, when the estimate $\widehat{f(x_i)}$ is high for a small change in y_i, or in other words, when model complexity is high. Thus, the previous equation can be rewritten as
  $$True\ error\ =\ empiral\ train\ error\ +\ small\ constant + \Omega(model\ complexity) \quad (22)$$
- This implies that true error cannot be reduced only by reducing the train error(maybe, even reduce to 0), but also by reducing the quantity $\frac{\sigma^2}{n}\sum_{i=1}^{n}\frac{\partial\widehat{f(x_i)}}{\partial y_i}$, or in other words the model complexity. This process is called $regularization$.
- **L2 regularization:** $L2$ regularization is given by the formula $\widetilde{\mathcal{L}(w)} = \mathcal{L}(w) + \frac{\alpha}{2}||w||^2$ where $\mathcal{L}(w) = \frac{1}{m}\sum_{i=1}^{m}(y_i - \widehat{yi})^2$. Thus in order to minimize the loss, we also need to minimize the second term, involving the square of the weights. This term can be thought of a proxy for the model complexity $\Omega(w)$ in eq.(22). Note that if α is 0, then there's no regularization.
- Taking the derivative of the formula for $\widetilde{\mathcal{L}(w)}$, we get $\nabla\widetilde{\mathcal{L}(w)} = \nabla\mathcal{L}(w) + \alpha w$

- Now, the update rule becomes $w_{t+1} = w_t - \eta\nabla\mathcal{L}(w) + \eta\alpha w$, in the case of vanilla gradient descent.
- It can be proved that

$$\nabla\mathcal{L}(w) = H\left(w - w^*\right) \tag{23}$$

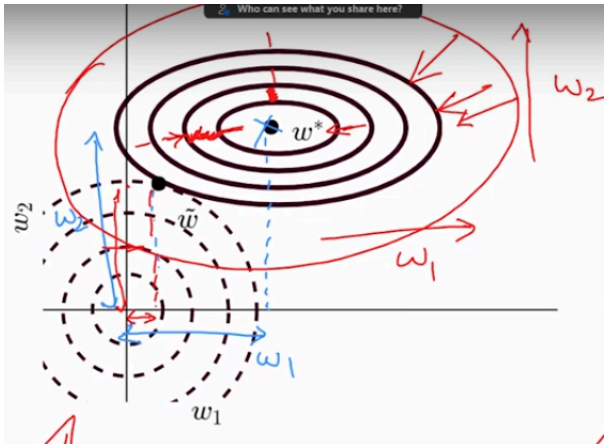
where H is the hessian matrix of the loss (second-order derivative of loss).

- Assume w^* is the optimal solution for $\mathcal{L}(w)$ and $\widetilde{w}$ is the optimal solution for $\widetilde{\mathcal{L}(w)}$. We now have $\nabla\mathcal{L}\left(w^*\right) = 0$ and $\nabla\widetilde{\mathcal{L}\left(\widetilde{w}\right)} = 0$. So, from the previous equations, it follows that $H\left(\widetilde{w} - w^*\right) + \alpha\widetilde{w} = 0$.
- Proceeding with the derivation in the lecture, at some point, we get $\widetilde{w} = QDQ^T w^*$, where D is a diagonal matrix of size $n\ x\ n$ given below

$$= \begin{bmatrix} \frac{\lambda_1}{\lambda_1+\alpha} & & & \\ & \frac{\lambda_2}{\lambda_2+\alpha} & & \\ & & \ddots & \\ & & & \frac{\lambda_n}{\lambda_n+\alpha} \end{bmatrix}$$

and Q is a transformation matrix of size $n\ x\ n$.

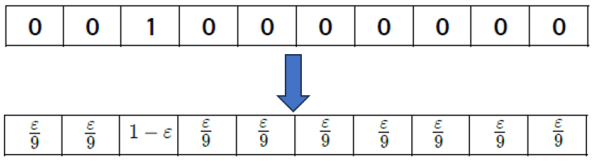
- Note that in the matrix D shown above, $\lambda_1,\ \lambda_1..\lambda_n$ are the eigen values of the Hessian matrix of the loss, H. If the eigen-values $\lambda_i \gg \alpha$,then the corresponding diagonal element becomes 1. If the eigen-values $\lambda_i \ll \alpha$,then the corresponding diagonal element becomes 0, which causes corresponding weights to *decrease/shrink.* This causes the the model complexity to reduce. Furthermore, the weight reduction happens more in those directions where loss declines slowly, as compared to the directions where loss declines sharply.



- $L2$ regularization penalizes all the parameters in the network equally, if all the eigen values of the Hessian matrix are equal.
- **Data augmentation:** Modification of the training samples to create new samples and adding them to the training set is called data augmentation. This is a very effective technique at regularizing the loss reduction, because with modified samples the model can't overfit as easy. Note that this is similar to $L2$ regularization.
- Data augmentation works well for image classification, object recognition, speech recognition, speech generation, text generation etc.
- **Adding noise to input :** Adding a noise to the input with a certain probability will make overfitting difficult, since *slightly different* inputs maps to the same output. THis evaluates to

$$E\left[\left(\tilde{y}-y\right)^2\right] = E\left[(\hat{y}-y)^2\right] + \sigma^2 \sum_{i=1}^{n} w_i^2$$

  and thus, equivalent to $L2$ regularization. In the above equation, $\tilde{y}$ is the actual output with the *noised* input and σ^2 is the variance of the noise (noise is assumed to be from the Gaussian $\mathcal{N}\left(0,\ \sigma^2\right)$.
- **Adding noise to output:** Adding a noise ϵ to the outputs (assuming that it's a classification problem) is pictorially represented as below.



In the above figure, the top part represents true y before adding noise, and bottom part represents true y after adding noise. Loss now evaluates to

$$\mathcal{L} = \sum_{i=1}^{n} y_i * log(\hat{y}) = \ \epsilon log\left(\widehat{q_1}\right) + (1-\epsilon) log\left(\widehat{q_2}\right) + \epsilon log\left(\widehat{q_3}\right) \ldots \ + \epsilon log\left(\widehat{q_n}\right)$$

  This is equivalent to $L2$ regularization, which has the form $L(w) + \epsilon\Omega(w)$
- **Early stopping:** Train/test error plotted over steps/iterations is very similar to that of train/test error plotted over model complexity. We would want stop training at a spot where the validation/test error (read curve) starts increasing

p is called patience parameter that indicates the number of consecutive steps/iterations where the validation error doesn't increase monotonically. k is the current step/iteraton number.

- From the eq.(23) in the section on $L2$ regularization above, we have

$$w_t = w_{t-1} - \eta \nabla \mathcal{L}(w) = w_{t-1} - \eta H\left(w_{t-1} - w^*\right) \tag{24}$$

where w^* and H are the optimum weight and the Hessian matrix for the *non-regularized loss* $\mathcal{L}(w)$ respectively.

- Proceeding with the derivation in the lecture, at some point, we get $w_t = QDQ^T w^*$, where D is a diagonal matrix $I - (I - \epsilon\Lambda)$. Size of D is $n \; x \; n$. This is similar to what we observed in the section on $L2$ regularization. In this case too, like with $L2$ regularization, the weight reduction happens more in those directions where loss declines slowly, as compared to the directions where loss declines sharply.
- **Ensemble:** Combining models (voting or average) reduces generalization error. Models could belong to the same type, might differ in hyperparameters, number of layers, features, training data (sampling with replacement).
- Assume there are k classifiers, each making an error ϵ_i drawn from $\mathcal{N}\left(0, \sigma^2\right)$. Further assume that the variance is represented as V and covariance among any two classifiers represented as C.

$$Mean\ square\ error\ (MSE) = \frac{1}{k}V + \frac{k-1}{k}C$$

- If the errors from classifiers are perfectly correlated, $V = C$. In that case, $MSE = V$. On the other hand, if the errors are perfectly uncorrelated, $MSE = \frac{1}{k}V$. Practically, the classifers would have a non-zero correlation and this results in a situation more desirable than either extremes.
- **Dropout:** However, when it comes to deep learning models, training with k models is prohibitively expensive and isn't practical. Hence, some neurons are dropped from the

network layers with a certain probability.

- By dropping out neurons in a network with n neurons, we can create 2^n other networks. However, these networks are not independently trained and then combined. Instead, each network is created by masking some connections in the original network. Corresponding connections in all networks 'share' the same weight and during the forward/backpropagation for each mini-batch, it gets updated only when the connection is present (not masked).
- If, in a network with n neurons, each neuron is retained with an 80% probability, there's a 64% probability that a weight (connection between two neurons, both of which must be retained) gets updated. Thus, if the network is trained using gradient descent run for 1000 iterations, the weights will gets updated 640 times.
- Neurons in the output layer of this network need to be scaled by the same probability (80% in our case).

Week7 - Activation functions and Initialization methods

- Training Neural Networks is a *Game of Gradients* (played using any of the existing gradient based approaches that we discussed)
- The gradient tells us the responsibility of a parameter towards the loss
- The gradient with respect to a parameter is proportional to the input to the parameters as shown below

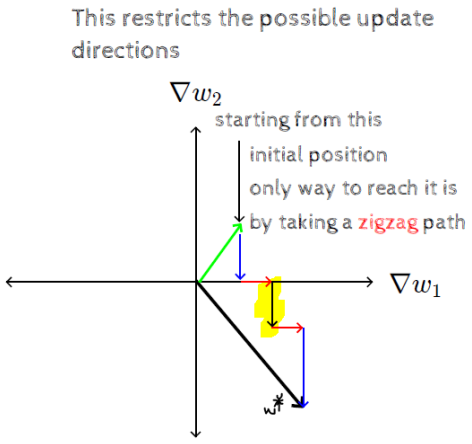
$$\nabla w_i = \frac{\partial \mathcal{L}(w)}{\partial y} * \frac{\partial y}{\partial a_3} * \ldots * \frac{\partial a_1}{\partial w_1}$$

$$NOTE: \frac{\partial a_1}{\partial w_1} = h_{i-1} \ (input\ of\ the\ previous\ layer)$$

- Backpropagation algorithm existed since 70's, but was popularized in the context of deep learning in 1986 by the paper authored by Rumerhart et.al. Deep learning became popular during late 2000's, due to the availability of enormous data and compute power.
- In unsupervised pre-training, we'll try and reconstruct the input x after passing it through a single hidden layer. The loss in this case can be represented as $min\ \frac{1}{m}\sum_{i=1}^{m}\sum_{j=1}^{n}\left(\widehat{x_{ij}} - x_{ij}\right)^2$, where n is the number of input dimensions and m is the number of training examples.
- Note that, in this case, the hidden layer applies the transformation $h_1 = \sigma(Wx)$ and the output layer applies the transformation $Wh_1 + b$. If we're able to find W such that we're able to reconstruct the input x with zero loss at the output layer, we've obtained

a good initialization for the weight W.

- Repeat this for the next layer of a deep network. Note that the input for the next layer is the output from the previous layer. THis helps in initializing weights across all layers.
- Optimization is done on training data, and regularization occurs on test/validation data.
- Remember, if the network is sufficiently complex (#weights), it'll drive down the loss to zero even without pre-training. This is also what universal approximation theorem suggests.
- Unsupervised pre-training is equivalent to minimizing the second part before minimising the first part in the regularized loss equation, $L(\theta) + \Omega(\theta)$.
- If all the activation functions in a network were replaced with linear functions (or removed), the output simply becomes a linear function. In other words, the power of network comes from the non-linearity provided by its activation functions.
- The derivative/gradient of a sigmoid function is given by $\sigma(x)(1 - \sigma(x))$ and vanishes at its saturation points when the input x is higher or lesser than ≈ 5. Note that $\sigma(x) = \sigma\left(\sum_{i=1}^{n} w_i x_i\right)$ and hence if the weights are reasonably high, the sigmoid will saturate. Specifically, this will happen as the number of neurons are more, since even though the individual weights w_i are small, the accumulated value could be high.
- Sigmoids are not zero-centered and this causes an issue during backpropagation. Note that the gradients with respect to weights are decided by the activation outputs of the previous layer, both of which are positive by definition (of sigmoid). Thus, the gradients with respect to individual weights at a specific layer will all be positive, or will all be negative. This is very limiting in terms of weight updates, and forces gradient descent to take zig-zag updates so as to reach the optimal w^* as shown in the figure.

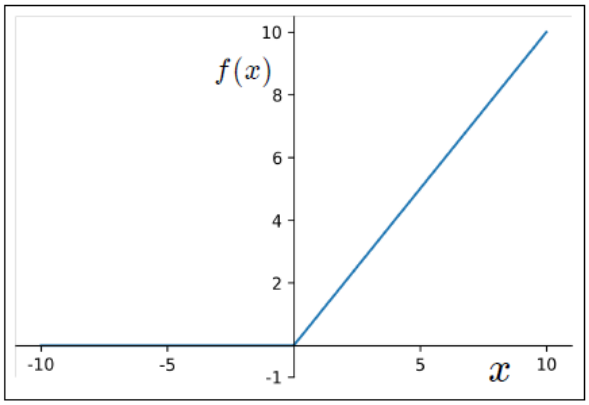


This zig-zag pattern of movement takes a long time to reach the optimum value of w. Also note that sigmoids are computationally expensive.

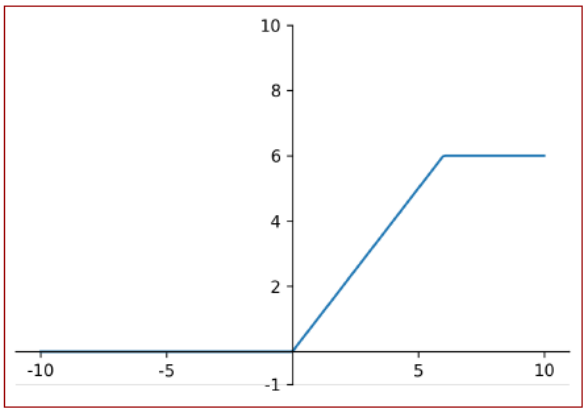
- Tanh is a popular activation function given by $f(x) = tanh(x)$. It has the advantage

that it's zero-centered, but saturates around $x = 2$ and hence not preferred.

- ReLU is a very popular activation function and given by $f(x) = max(0, x)$.



There also exists some closely related functions that could used instead of ReLU. For example, ReLU6 is given by $f(x) = max(0, x) - max(0, x - 6)$. This function resembles a sigmoid and saturates at $x = 6$.

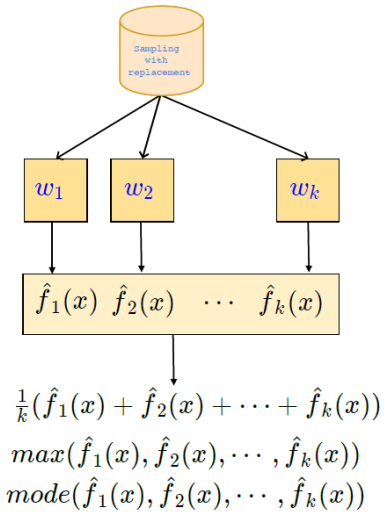


- Advantages of a ReLU is that it doesn't saturate in the positive region. Computationally, it's much more efficient than sigmoid.
- The main disadvantage of ReLU is that gradient vanishes in the negative direction, which is possible when the bias (or the weights) are negative. In this case, the neuron is referred to as *dead*, which also means that it'll remain un-updated during rest of the gradient descent and hence remain *dead* forever. This situation typically occurs in a ReLU when the learning rate is set to a high value, which might cause bias to update to a large negative value. Hence, it's advisable to set learning rate to 0.01, when ReLU is used.
- Leaky ReLU $(f(x) = max(0.1x,\ x))$ allows a slight leakage on the negative side, instead of saturating at all negative inputs. Parametric ReLU $(f(x) = max(\alpha x,\ x))$ uses a hyper-parameter α instead of hard-coding the coefficient of x to 0.1.
- Exponential Linear Unit (ELU) exponentially decays the output from activation function

exponentially as given by

$$f(x) = x \quad if \quad x > 0$$
$$= ae^x - 1 \quad if \quad x \leq 0$$

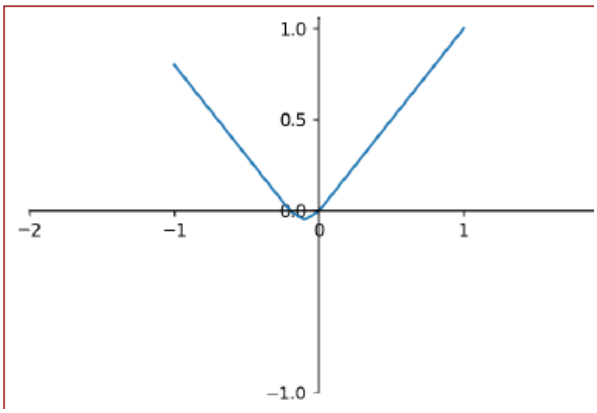
- **What is Maxout strategy?**
  When you do sampling with replacement, nearly 36% of training samples are duplicates. So, it's possible that the model could overfit. The model averaging (arithmetic/geometric mean, max or mode) strategy is used to draw the inference/prediction. This technique is also referred to as bagging.



- In the case of dropout strategy, sub-models are created by dropping each neuron in turn from the original network. Since there are an exponential number of sub-models in this case, each of them train only with certain parts of the training samples. Thus, not only the model doesn't overfit on the training samples, but requires higher updates so that learning occurs. It's not recommended to have higher learning rate, because that'll apply to the entire model. Hence, the ideal solution is to use $max()$ over all the neuron outputs during the forward propagation, so that the gradient flows through the nuerons that produce higher outputs. This activation is referred to as max-out and is represented by

$$f(x) = max(w_1 x + b_1, \ldots w_n x + b_n)$$

  Maxout activation generalizes ReLU or other variations of it. Thus, when there are only two neurons in the layer with $w_1 = 0,\ b_1 = 0,\ w_2 = 1,\ b_2 = 0$, the above equation essentially boils down to ReLU - $max(0,\ x)$.
  As another example, consider 4 neurons in a layer which produce the following outputs. $(0.5x,\ -0.5x,\ x,\ -x - 0.2)$. When maxout strategy is applied, it produces the following activation.



- To summarize, linear activation can be represented as

$$f(x) = \begin{cases} 1, & \text{if } x > 0 \\ 0, & x \leq 0 \end{cases}$$

ReLU is represented as

$$f(x) = \begin{cases} 1 \cdot x, & \text{if } x > 0 \\ 0, & x \leq 0 \end{cases}$$

Dropout can be represented as

$$\mu(x) = \begin{cases} 1 \cdot x, & p \\ 0 \cdot x, & 1 - p \end{cases}$$

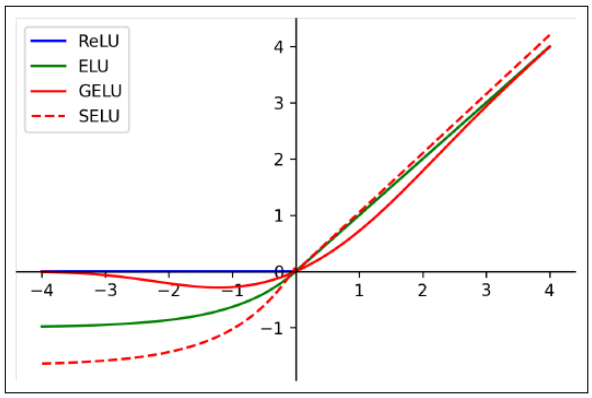
Generalizing this, we've another activation function called GELU

$$f(x) = m \cdot x$$
$$\text{where,} \qquad m \sim Bernoulli(\Phi(x))$$

Note that $0 \leqslant \phi(x) \leqslant 1$ and typically represented by a sigmoid or the CDF (Cumulative denisty function) of x, and in our case the pre-activation output a_{ij}. $Bernoulli$ outputs 1 with a probability $\phi(x)$, and 0 with a probability $1 - \phi(x)$. Expected value of the above activation is given by $E(f(x)) \ = \ E(m.\, x)$ calculated further below.

$$E(m.\, x) = \phi(x).1.x + (1 - \phi(x)).0.x = \phi(x).\, x = P(X \leqslant x).\, x = \sigma(1.702x).\, x$$

- SELU is yet another activation function similar to ReLU/GELU, but with zero-centering.
- Following figure shows a few activation functions we've learnt until now



- SWISH activation function generalizes GELU and given by $\sigma(\beta x).\, x$. When $\beta = 1.702$, we get GELU. When $\beta = 1$, we get SILU.