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.

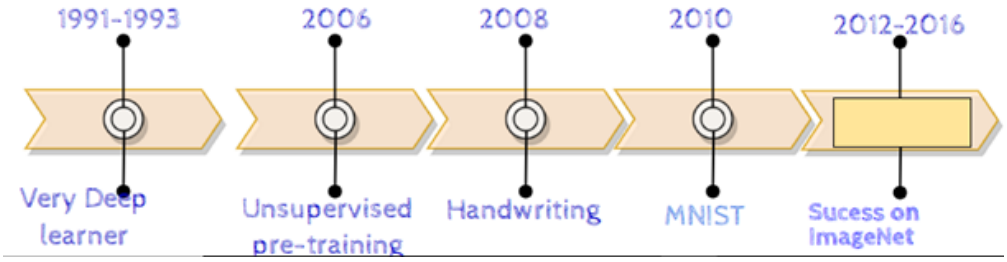


Figure 1: Deep neural networks over the decades.

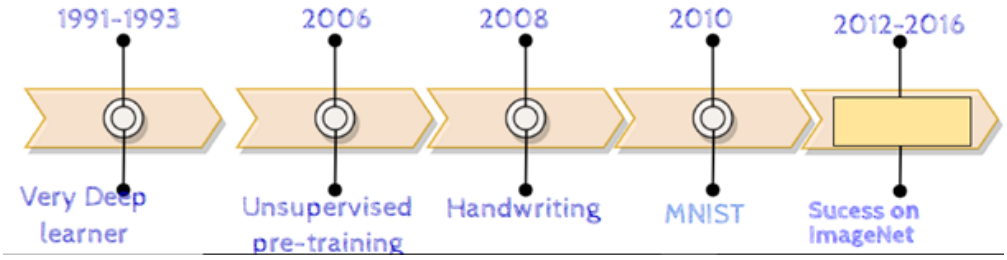


Figure 2: Evolution of convolutional neural networks.

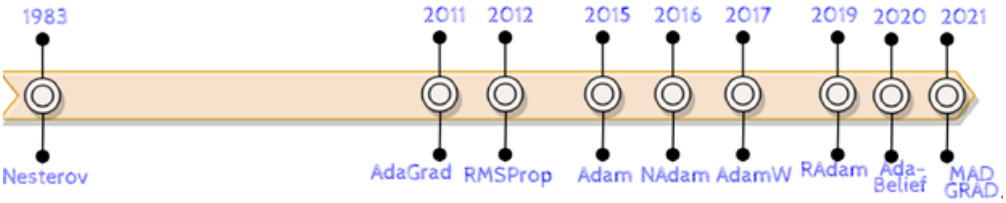


Figure 3: Faster convergence

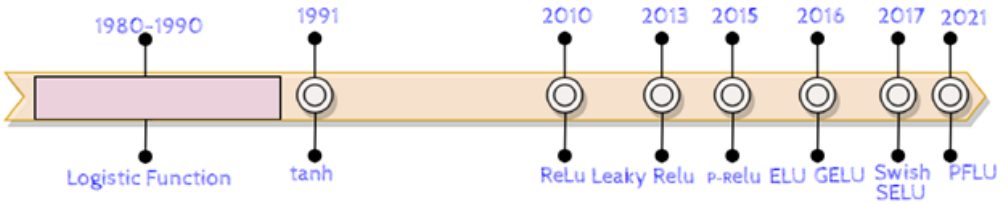


Figure 4: Better activation functions

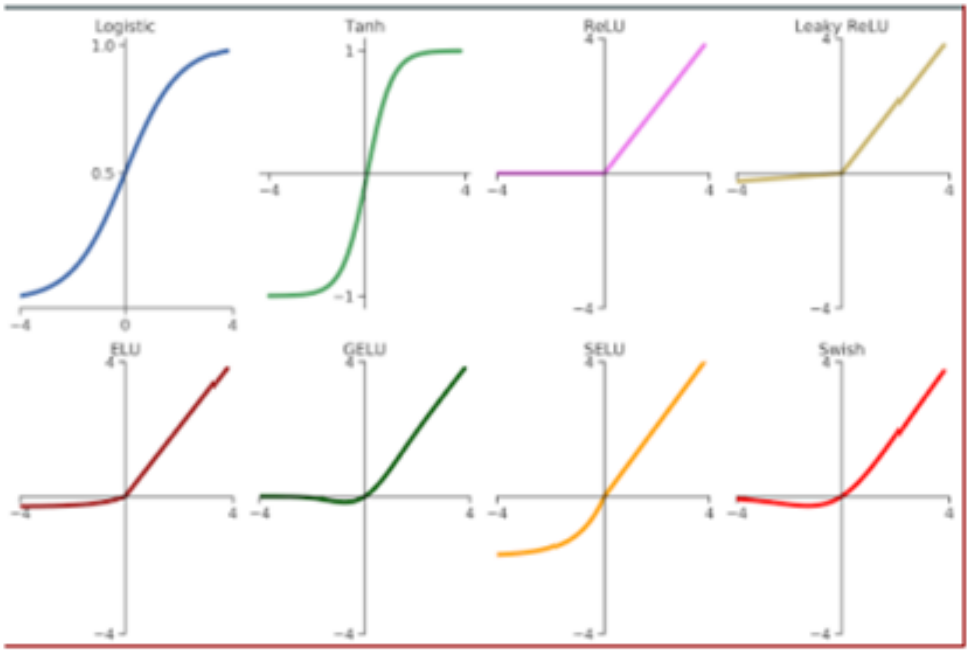


Figure 5: Various convergence functions

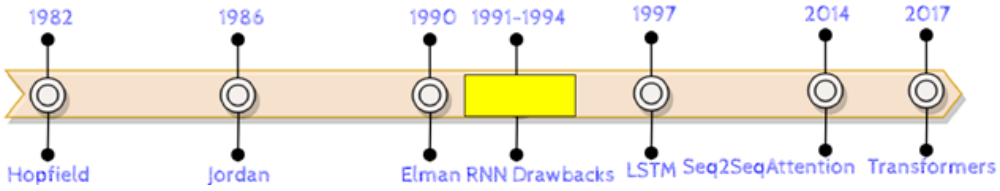


Figure 6: Sequences (sound/video)

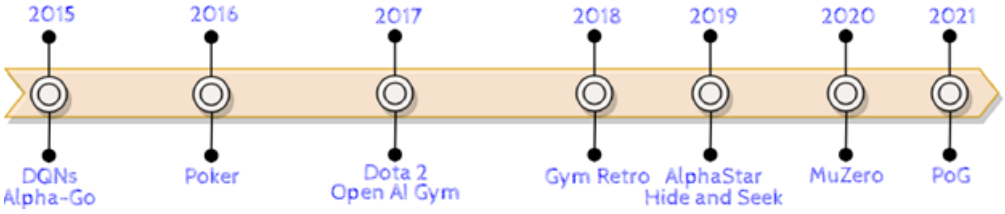


Figure 7: Games

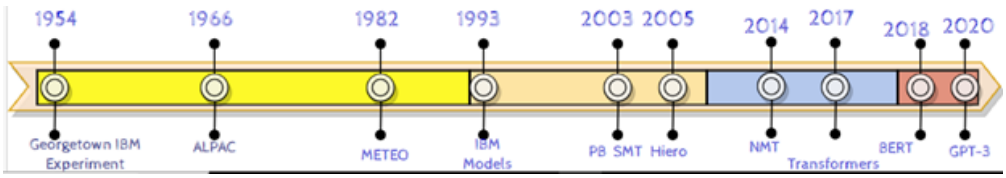


Figure 8: Rise of transformers

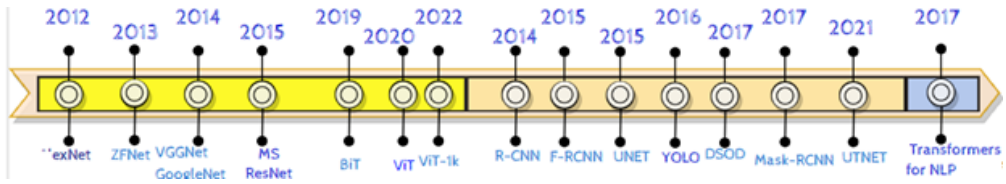


Figure 9: Transformers everywhere!

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

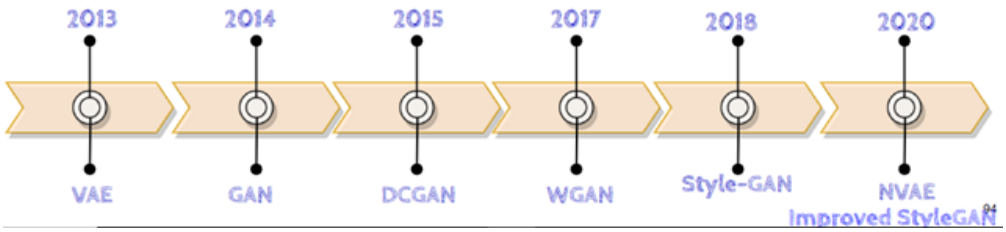


Figure 10: Generative models

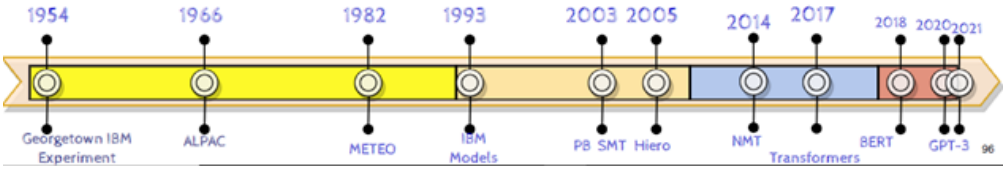


Figure 11: Language-Vision

- 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.

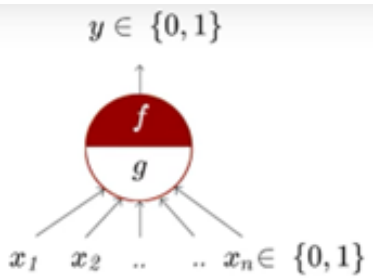


Figure 12: McCulloch Pitts Neuron

  - 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.

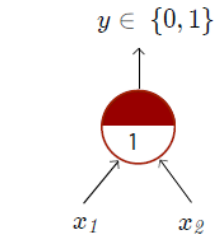


Figure 13: McCulloch Pitts neuron representing an OR logic with 2 inputs

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)

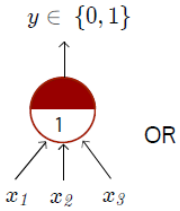
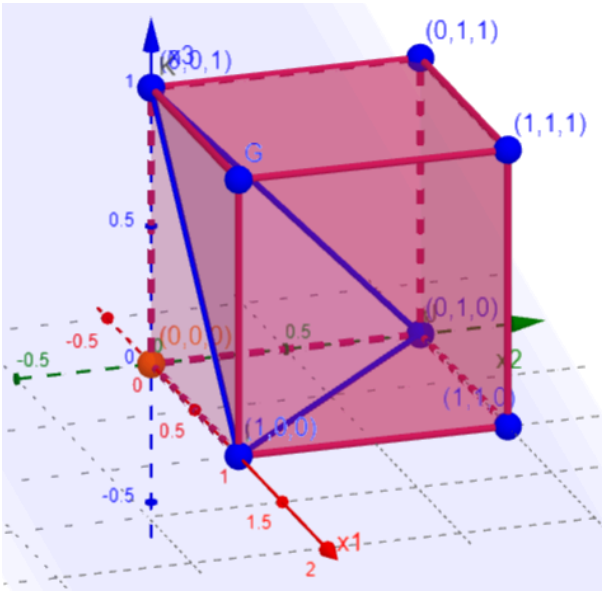


Figure 14: Neuron that represents OR logic with 3 inputs

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

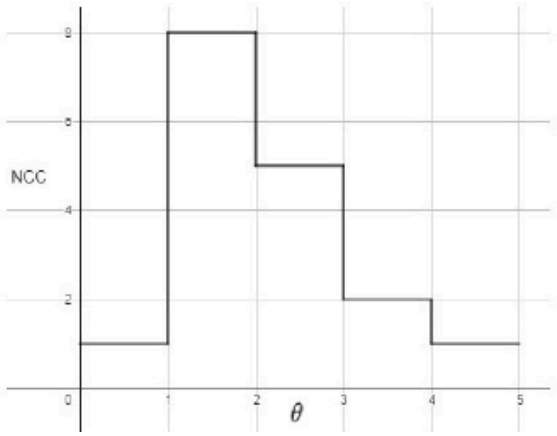
$$w^T x > 0 \implies -0.5x_1 - 0.5x_2 > 0 \implies -0.5x_1 > 0.5x_2$$

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

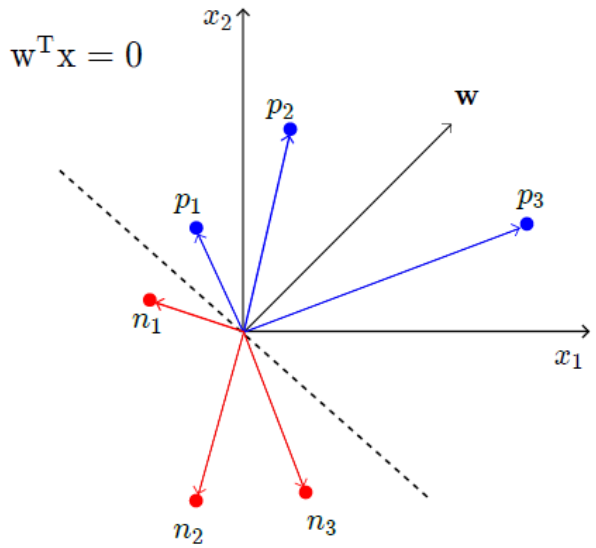


Figure 17: Weight vector casts $90°$ with data points on the dividing line $w^T x = 0$

- 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

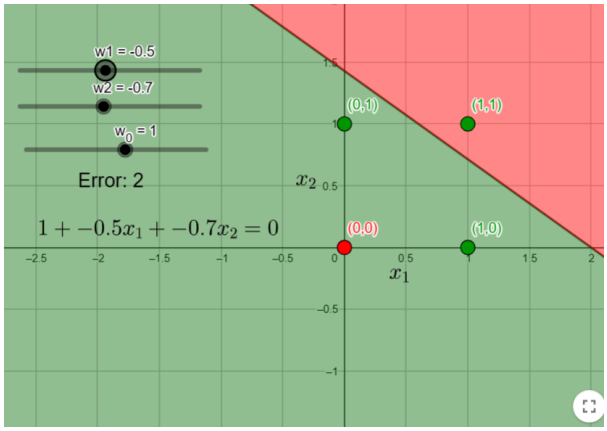
Figure 18: NAND truth table

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_true	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.					

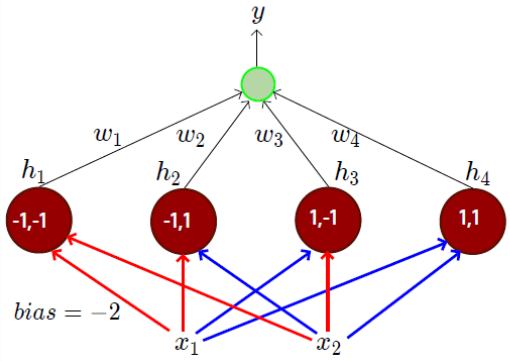
Week2

- 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.



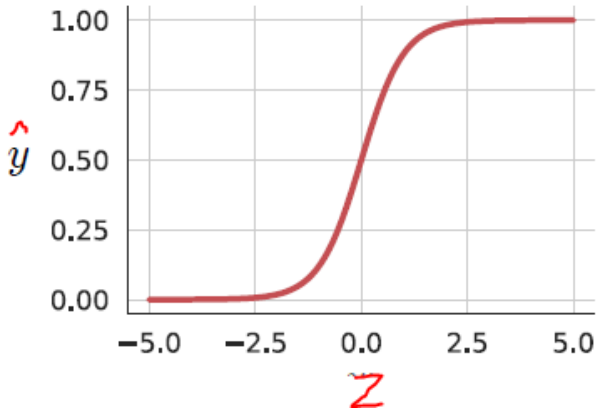
Let w_0 be the bias output of the neuron (i.e., it will fire if $\sum_{i=1}^{4} w_i h_i \geq w_0$

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

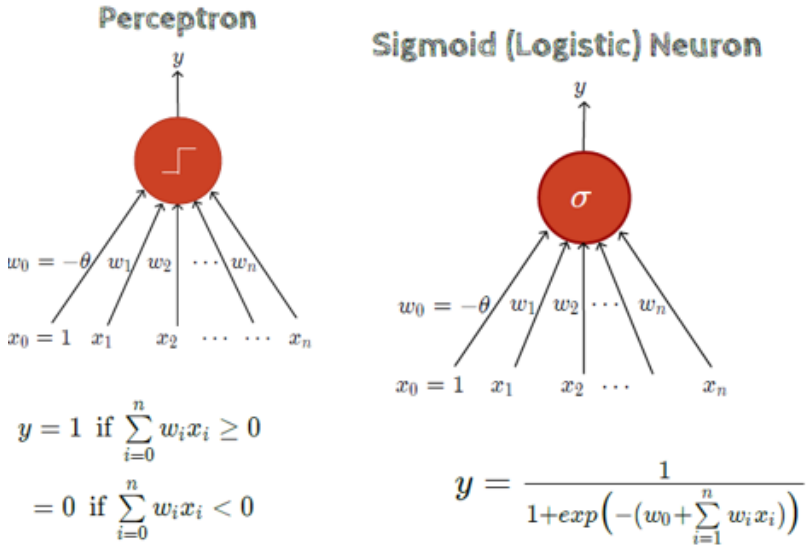
- A hidden layer of m neurons is capable of implementing 2^m functions.
- The issue is that for a large value of n, the MLP would be unmanageable. As an example, creating a hidden layer with 2^{100} perceptrons in the case of a 100-feature input is near to impossible.
- Perceptron uses a step function 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.



- 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 $w_0 + w_1 x_1 + w_2 x_2 = 0$ 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 $\widehat{y}$). Ideally, we want the y and $\widehat{y}$ to be equal. Or, rather the difference between these two quantities to be minimum. We define $\mathcal{L}(w) = \sum_{i=1}^{n} (\widehat{y}_i - y_i)^2$ 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 ($\mathcal{L}(w) = \sum_{i=1}^{n} |\widehat{y} - y_i|$), 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.

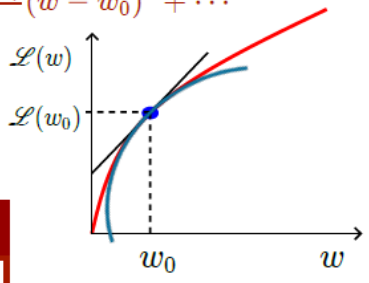
$$\mathscr{L}(w) = \mathscr{L}(w_0) + \frac{\mathscr{L}'(w_0)}{1!}(w - w_0) + \frac{\mathscr{L}''(w_0)}{2!}(w - w_0)^2 + \frac{\mathscr{L}'''(w_0)}{3!}(w - w_0)^3 + \cdots$$

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,

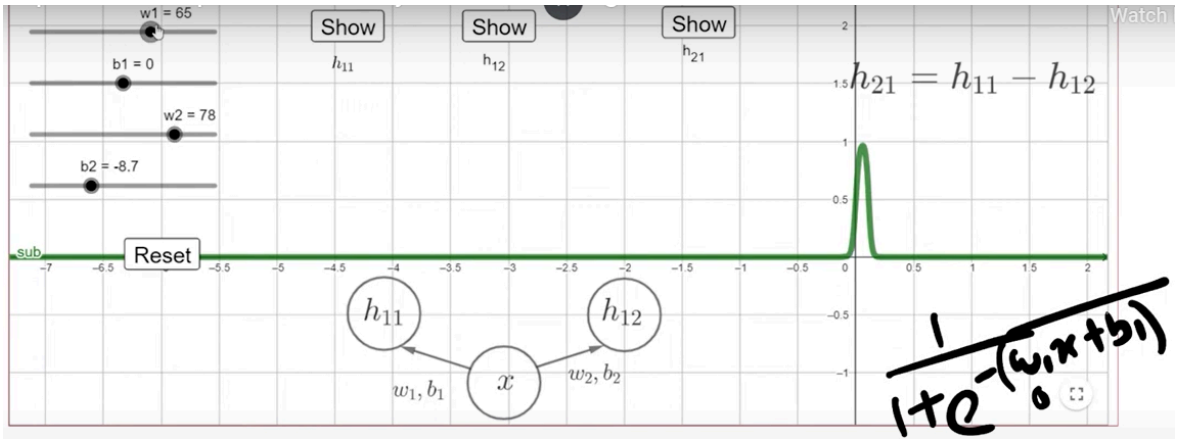
$$\nabla w\ =\ \sum_{i=1}^{N} (f(x_i)\ -\ y_i) * f(x_i) * (1\ -\ f(x_i)) * x_i \tag{9}$$

$$and$$

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

- 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,

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

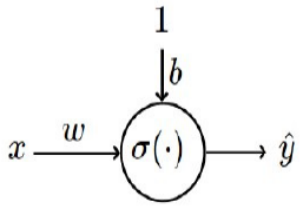
From the equation for $\hat{y}$,

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

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

Q2.

Consider a sigmoid neuron shown below. The input to the neuron is a real number. Suppose that input $x = 10$, output $y = 1$ and the parameters, $w = 0.1, b = 0.1$. Update the parameters once using GD with η=1. Enter the sum of updated parameter values. The loss function is $L = \frac{1}{2}(y - \hat{y})^2$.

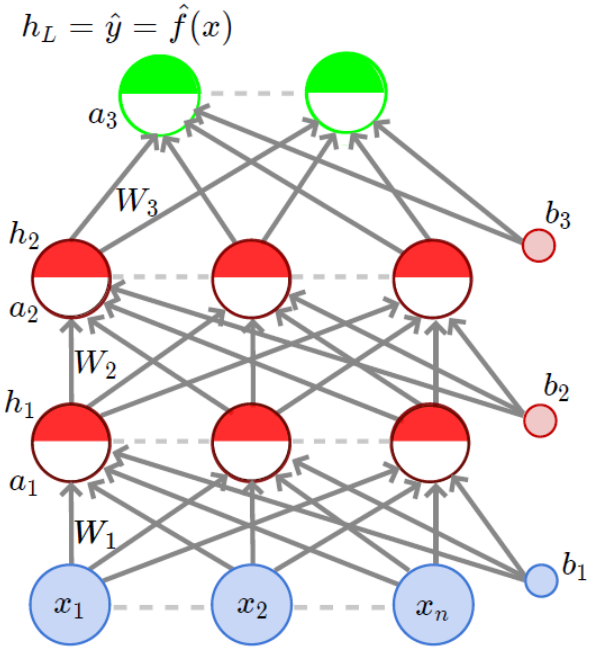


Answer:

```
import math

x = 10
y = 1
w = 0.1
b = 0.1
eta = 1

# Step 1: Compute the output y_hat
z = w * x + b
y_hat = 1 / (1 + math.exp(-z))

# Step 2: Compute the loss (not needed for further calculations, so skipping)
# Step 3: Compute the gradients
dL_dy_hat = y_hat - y
dy_hat_dz = y_hat * (1 - y_hat)
dL_dz = dL_dy_hat * dy_hat_dz

dL_dw = dL_dz * x
dL_db = dL_dz * 1

# Step 4: Update the parameters
w_new = w - eta * dL_dw
b_new = b - eta * dL_db

# Step 5: Sum of updated parameter values
sum_updated_parameters = w_new + b_new
```

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.

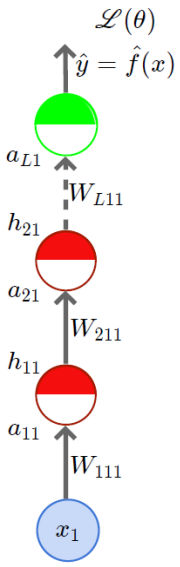


Figure 22: A very thin network

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

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

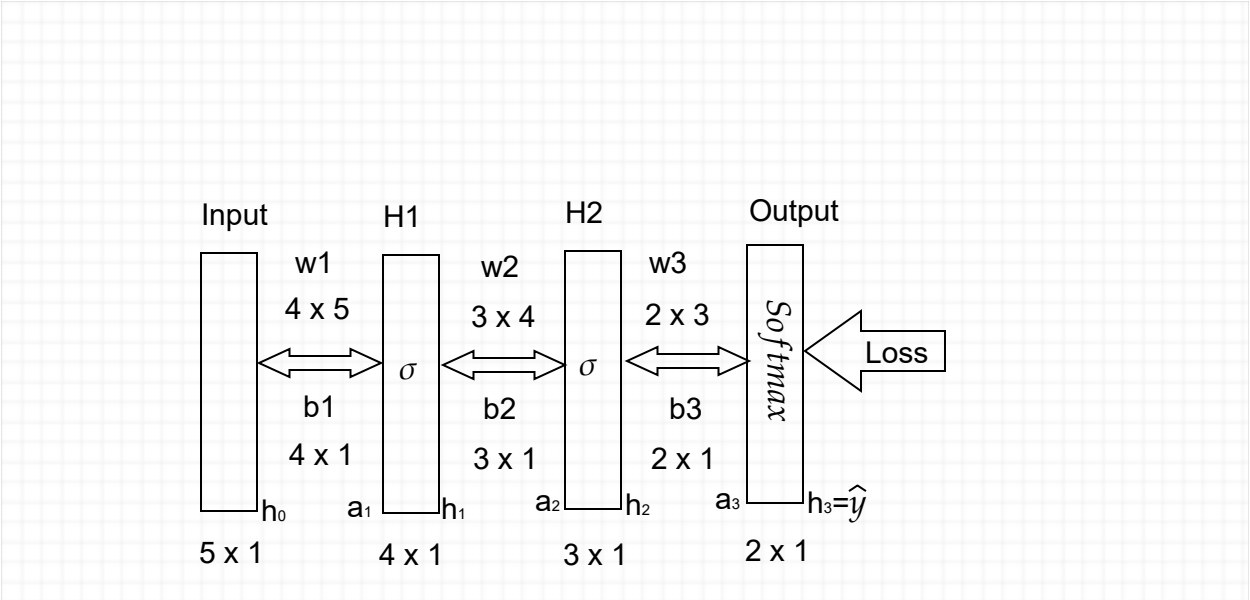
$$\frac{\partial \mathcal{L}(\theta)}{\partial W_{111}} = \frac{\partial \mathcal{L}(\theta)}{\partial h_{11}} \frac{\partial h_{11}}{\partial W_{111}}$$ (just compressing the chain rule)

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.
  - $a_1 = w_1 @ h_0 + b_1$
  - $h_1 = \sigma(a_1) = \frac{1}{1 + e^{-a_{1i}}}$
  - $a_2 = w_2 @ h_1 + b_2$
  - $h_2 = \sigma(a_2) = \frac{1}{1 + e^{-a_1}}$
  - $a_3 = w_3 @ h_2 + b_3$

- $h_3 = Softmax(a_3) = \frac{e^{a_{3i}}}{e^{a_{31}} + e^{a_{32}}}$

- 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 $A = \left[w_3^T \,@\, (\hat{y} - y) \odot h_2(1 - h_2)\right]$, first part in the above equation.

- $\nabla_{w_1}^{\mathfrak{L}} = \left[w_2^T \,@\, A \odot h_1(1 - h_1)\right] @\, h_0^T$
- $\nabla_{b_1}^{\mathfrak{L}} = w_2^T \,@\, A \odot h_1(1 - h_1)$

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:

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

  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.

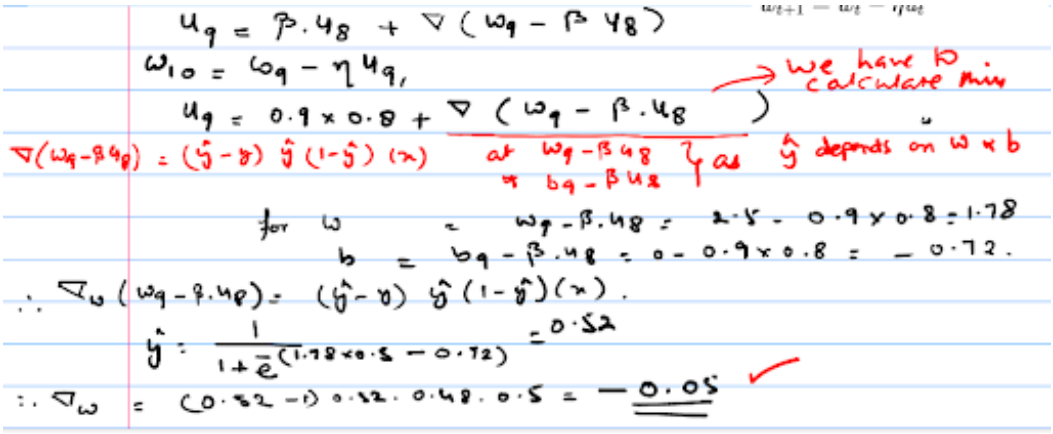
Problems

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 $\{x_1,\ x_2,\ x_3,\ x_4\}$ that feeds into a sigmoid nueron, we can represent the associated 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 $\sum_{i=1}^{n} (f(x) - y) * f(x)(1 - f(x)) * x_i^2$
- 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. In other

words, decay the learning rate for parameters in proportion to their update history.

- 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, slower does v build up, and hence slower does the effective learning rate η decay due to the self-correction mechanism $\frac{\eta_0}{\sqrt{v_t + \epsilon}}$, causing smaller weight updates. On the other hand, the denser the feature is, faster does v build up, and hence faster does the effective learning rate decay due to the self-correction mechanism mentioned above, causing larger weight updates.

- The below graph is plotted for sparse features. Note that ∇w 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.

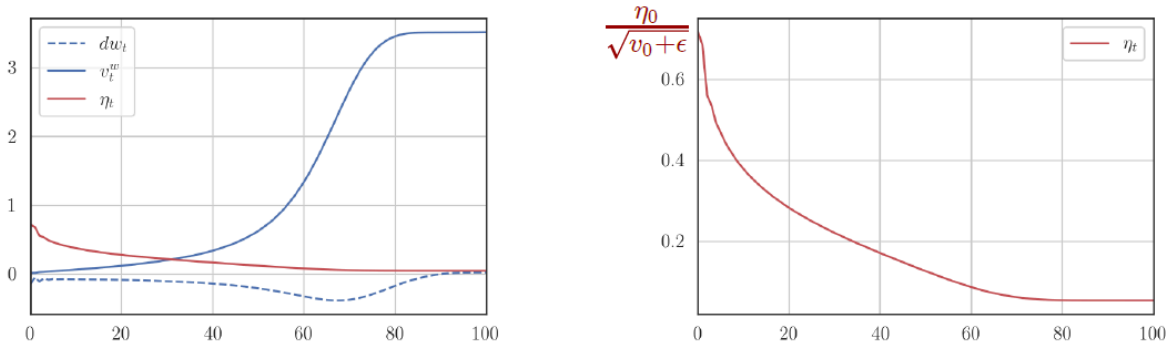


Figure 23: AdaGrad - Sparse feature

- The below graph is plotted for dense features. Note that the scale of the graph is much higher due to the rapid growth of v_t and hence the ∇w plot is not visible. The effective learning rate (shown on the right side) decays rapidly through iterations.

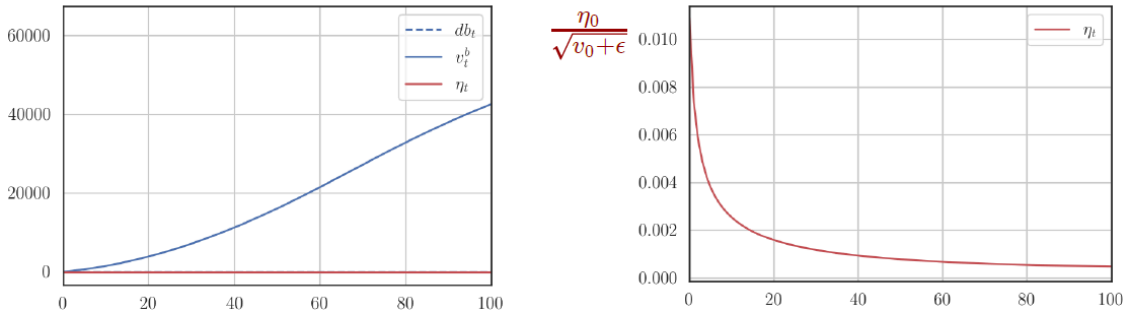


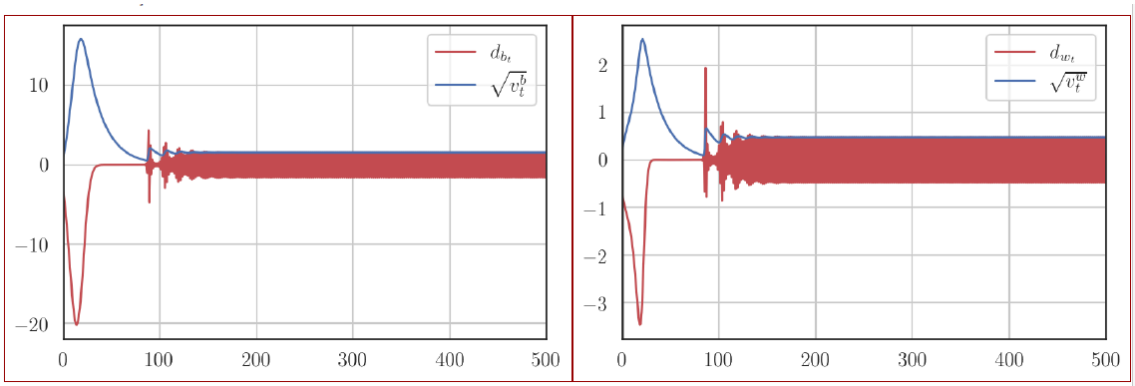
Figure 24: AdaGrad - Dense feature

- 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:

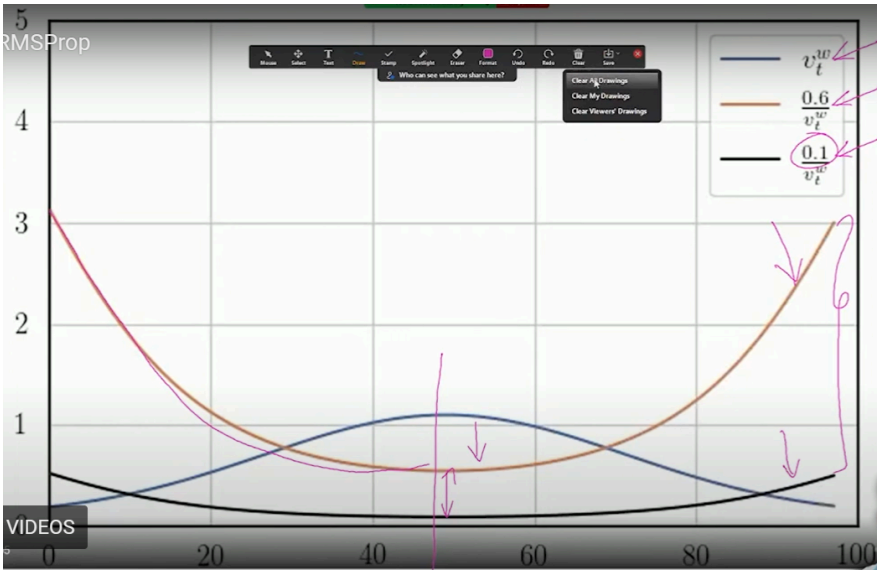
$$v_t = \beta v_{t-1} + (1-\beta)(\nabla w_t)^2$$
$$w_{t+1} = w_t - \frac{\eta_0}{\sqrt{v_t+\epsilon}} * \nabla w_t \quad (16)$$

The above rule ensures that, for dense features, learning rate decays slowly, instead of rapidly (as in AdaGrad). Note that the typical value of β is 0.9, which ensures that the decay rate of the past gradients is high.

- On the downside, RMSprop causes the learning rate 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 the previous velocities gets progressively multiplied with the β, β^2, β^3... β^k, and hence decays during each iteration. 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. However, this 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, $v_t = 0.1$
$\eta_t = \frac{0.6}{1.25} = 0.48$	$\eta_t = \frac{0.6}{0.1} = 6$
$\eta_t = \frac{0.1}{1.25} = 0.08$	$\eta_t = \frac{0.1}{0.1} = 1$

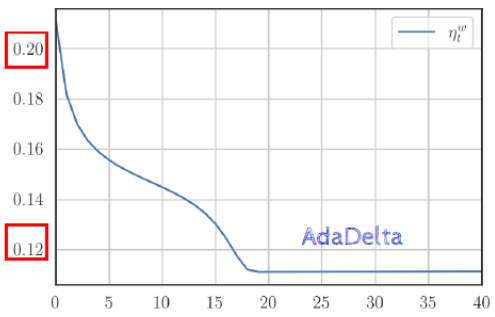
$\eta_t = 0.08$ is better in a steep region and $\eta_t = 6$ is better in a gentle region. Therefore, we wish the numerator also to change wrt gradient/slope

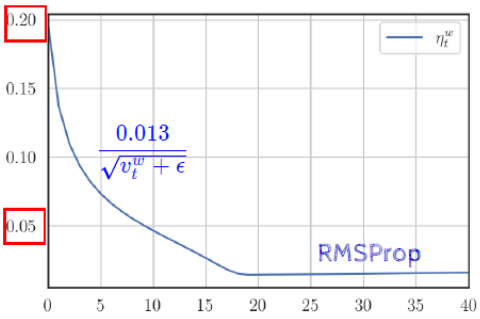
- In AdaDelta, the numerator η_0 in the learning rate computation is replaced a history of $-\sqrt{u_{t-1} + \epsilon}$, where $u_t = \beta u_{t-1} + (1 - \beta)\left(\Delta w_t^2\right)$
- The update rule for AdaDelta is as follows:

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

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. Performing bias correction on the velocity reduces the learning rate and will help speed up convergence, but lack of bias correction will not prevent the algorithm from convergence.

- $\widehat{v_t}$ 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
- In general, 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 $L_1 = 1$.

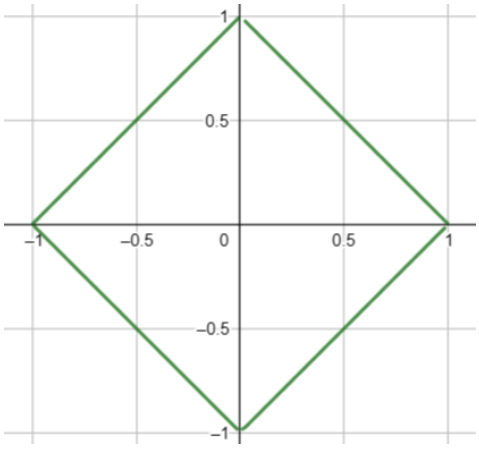


Figure 25: L_1 norm for all points on this unit square = 1

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 effective learning rate increases whenever zero inputs are encountered, due to the exponentially decaying velocity.

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

- When $p = \infty$, $L_p = max(x_1, x_2 ... x_n)$. Using the max norm (instead of $L2$ in the update rule for Adam in eq.(18), we have the following equations for velocity vector and the weight update.

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

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 effective learning rate doesn't get modified when encountering zero inputs.

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

- The update rule for AdaMax is as follows:

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

- 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 a model once. 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

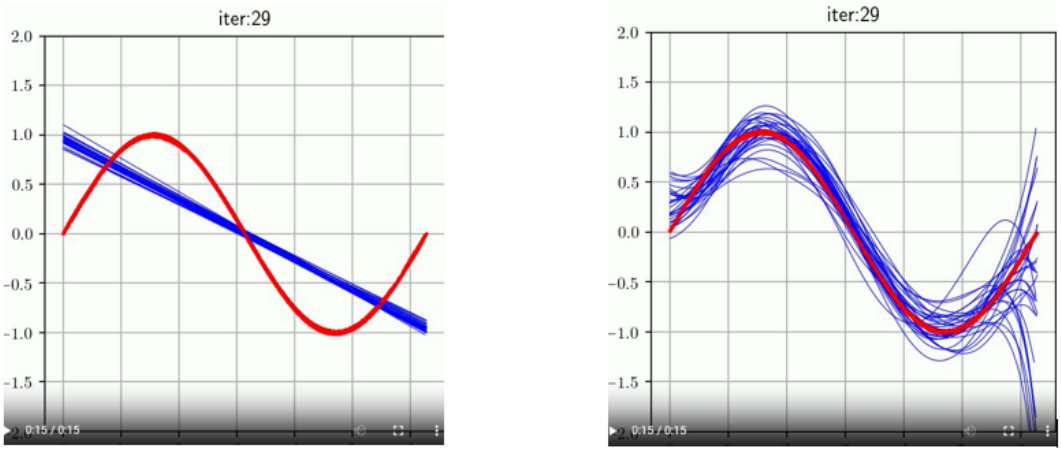


Figure 26: Fitting the samples to a simple model (left) and complex model (right)

- .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, mathmatically represented as $E\left[\widehat{f(x)}\right] - f(x)$. Simple models have high bias and complex models have low bias.
- $Variance$ is defined as the mean squared error between the average(expectation) from k models and each of the k models, mathematically represented as $E\left[\left(\widehat{f(x)} - E\left[\widehat{f(x)}\right]\right)^2\right]$. Simple models have low variance and complex models have high variance.
- Typically, when the model have a low bias, it has a high variance and vice-versa. 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 test/validation error between the true value and prediction, for a given x and is related to the bias and variance by the following

relationship.

$Test/validation\ error\ =\ Bias^2 + Variance\ +\ \sigma^2$ (The last quantity is called the irreducible error, due to noise in the input).

- 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)

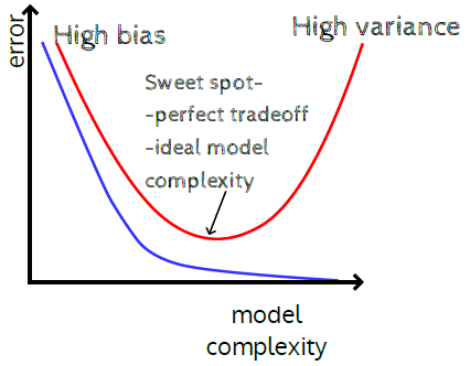


Figure 27: Train and test errors.

- Model with high bias produce high training error and high validation error, whereas model with high variance produce low training error and high validation error.
- Bias and variance can be both driven down to zero if the underlying distribution of the samples are known and also the samples are not corrupted by noise
- Assuming that there are $(n+m)$ samples, n of which are training and m are test samples, empirical train error is given by the formula $\frac{1}{n}\sum_{i=1}^{n}\left(\widehat{f(x_i)} - y_i\right)$ and empirical test error is given by the formula $\frac{1}{m}\sum_{i=n+1}^{m+n}\left(\widehat{f(x_i)} - y_i\right)$.
- It can be proved mathematically that

  $True\ error\ =\ empiral\ \textbf{test}\ error\ +\ small\ constant.$ and

  $True\ error\ =\ empiral\ \textbf{train}\ error\ +\ small\ constant + E\left[\epsilon\left(\widehat{f(x)} - f(x)\right)\right]$, 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 empirical 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). α can take any value between 0 to ∞. If the value of 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

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

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

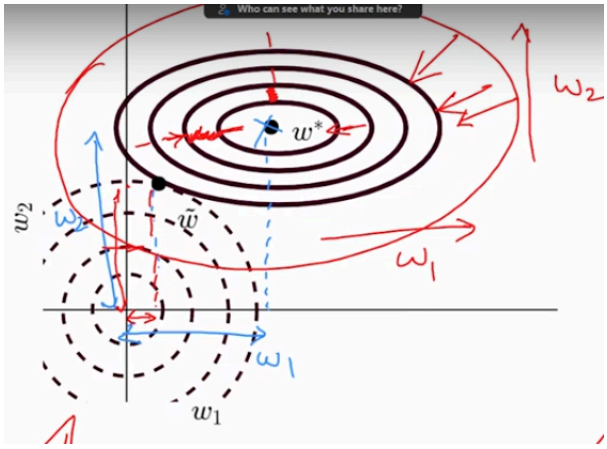
- Assume w^* is the optimal solution for $\mathcal{L}(w)$ and $\widetilde{w}$ is the optimal solution for $\widetilde{\mathcal{L}(w)}$. We now have $\nabla\mathcal{L}\left(w^*\right) = 0$ and $\nabla\widetilde{\mathcal{L}\left(\widetilde{w}\right)} = 0$. So, from the previous equations, it follows that $H\left(\widetilde{w} - w^*\right) + \alpha\widetilde{w} = 0$.
- 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 model complexity to reduce. Furthermore, the weight reduction happens more (w_1) in those directions where loss declines slowly, as compared to the directions where loss declines sharply (w_2).

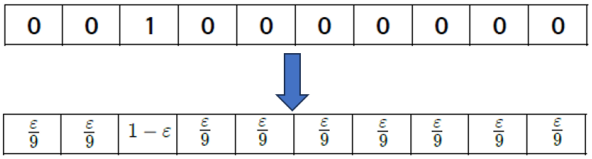


- $L2$ regularization penalizes all the parameters in the network equally, if all the eigen values of the Hessian matrix are equal.
- **Data augmentation:** Modification of the training samples to create new samples and adding them to the training set is called data augmentation. This is a very effective 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 must map 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 predicted output with the *noise-laden* input and σ^2 is the variance of the noise (noise is assumed to be from the Gaussian $\mathrm{N}\left(0,\ \sigma^2\right)$.

- **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

$$\mathcal{L} = \sum_{i=1}^{n} p_i * log(q_i) = \epsilon log\left(\widehat{q_1}\right) + (1-\epsilon) log\left(\widehat{q_2}\right) + \epsilon log\left(\widehat{q_3}\right) ... + \epsilon log\left(\widehat{q_n}\right)$$

where p_i is the true probability, and q_i is the predicted probability.

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 to 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. If the loss is increasing monotonically in the last p steps, stop the training and roll back all updates done in the last p steps.

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

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

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 $\mathrm{N}\left(0, \sigma^2\right)$. Further assume that the variance among the classifiers is represented as V and covariance among any two classifiers represented as C. Then, it can be proven that

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

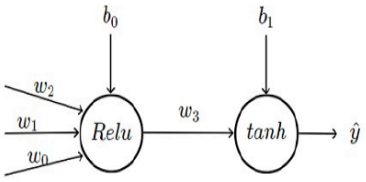
- If the errors from classifiers are perfectly correlated, $V = C$. In that case, $MSE = V$, which implies that bagging has no benefit. On the other hand, if the errors are perfectly uncorrelated, $MSE = \frac{1}{k}V$. Practically, the classifers would have a non-zero correlation and this results in 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.
- During test time, output of the neurons in this network will be scaled by the same probability (80% in our case) as was used during the training time.

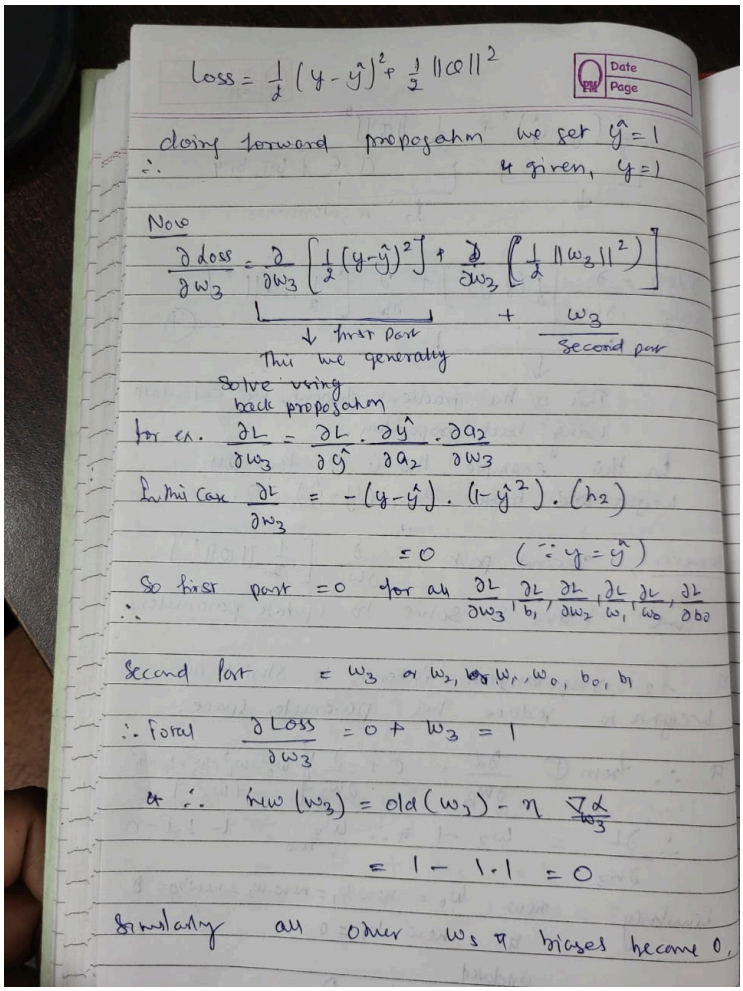
Problems

Consider a model shown below. All weights and biases except b_1 are initialized to 1 and b_1 is initialized to 10. Suppose that the input is all ones and the true label $y = 1$. Moreover, the model uses the following loss function with regularization,

$$\mathcal{L}(\theta) = \frac{1}{2}(y - \hat{y})^2 + \frac{1}{2}\|\theta\|_2^2$$

where, $\theta \in \{w_i, b_i\}$ (consider regularizing bias as well, though not followed in practice). The model parameters were updated using SGD with $\eta = 1$. Update the parameters by running the model for one iteration and enter the sum of updated parameters. (If your answer is 15.3325, then enter it as 15.33)





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

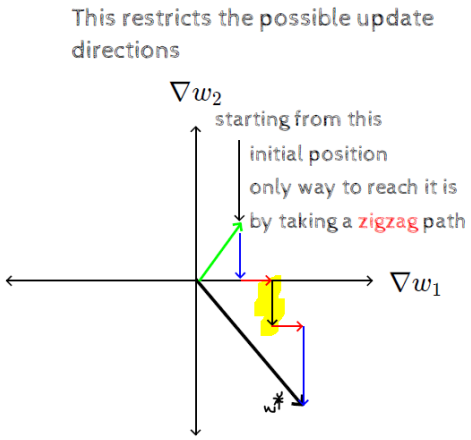
- The gradient with respect to a parameter is proportional to the input as shown below

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

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

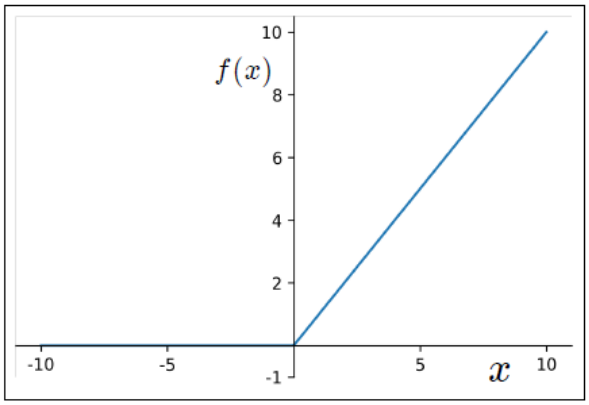
- Backpropagation algorithm existed since 70's, but was popularized in the context of deep learning in 1986 by the paper authored by Rumelhart et.al. Deep learning became popular during late 2000's, due to the availability of enormous amounts of data and compute power.
- In unsupervised pre-training, 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 than $\approx +5$ 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.

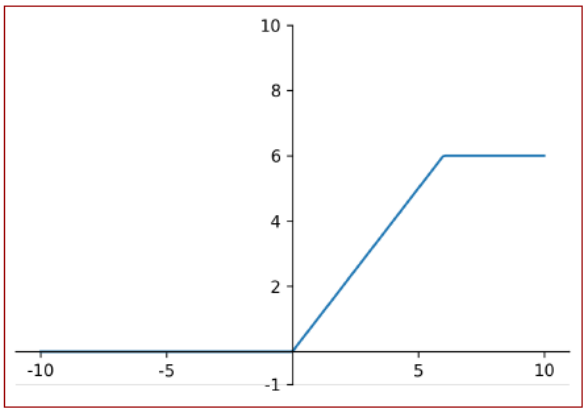


This zig-zag pattern of movement takes a long time to reach the optimum value of w. Also note that sigmoids are computationally expensive.

- Tanh is a popular activation function $f(x) = tanh(x)$. 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 $f(x) = max(0, x)$.



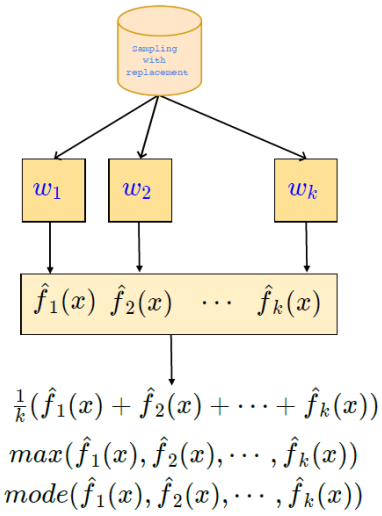
There also exists some functions similar to ReLU and hence could be used instead. For example, ReLU6 is given by $f(x) = max(0, x) - max(0, x - 6)$. This function resembles a sigmoid and saturates at $x = 6$.



- 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) is negative. In this case, the neuron is said to be $dead$, 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, whenever 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

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

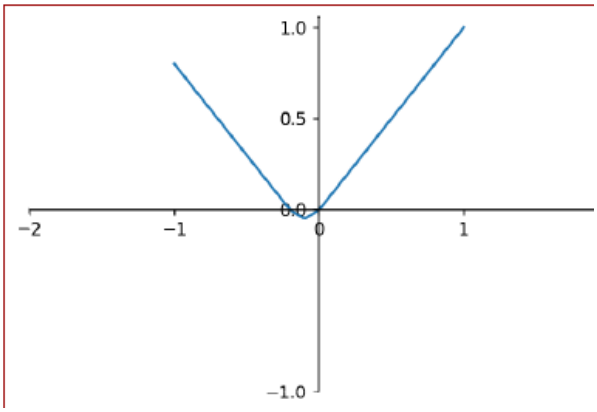
- **What is Maxout strategy?**
  When you do sampling with replacement, nearly 36% of training samples are duplicates. So, it's possible that the model could overfit. The model averaging (arithmetic/geometric mean, max or mode) strategy is used to draw the inference/prediction. This technique is also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

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

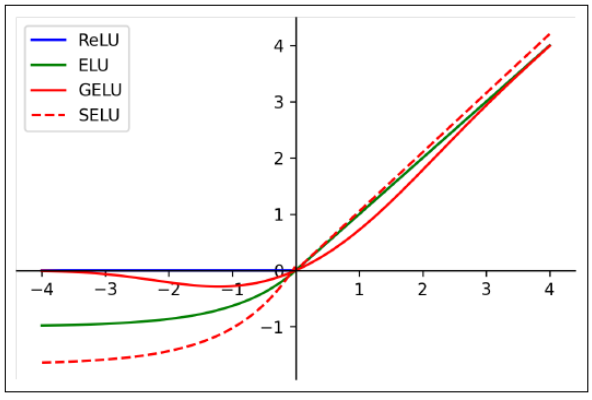
Generalizing this, we've another activation function called GELU

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

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. SELU ensures zero mean and unit variance outputs, provided the inputs are normalized and the network weights are initialized correctly.
- 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.

Week8 - Convolutional Neural Networks (CNN)

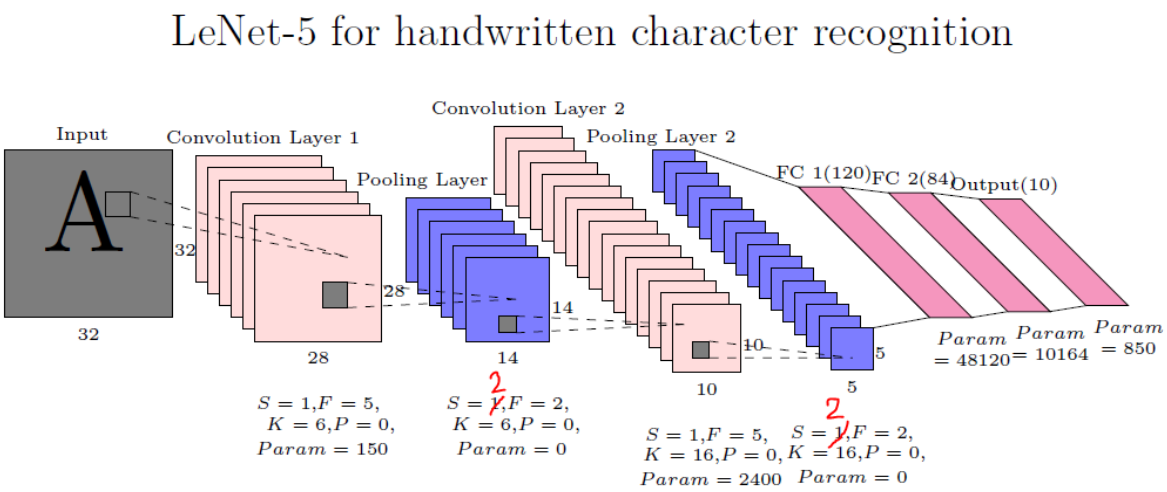
- Weighted average of a set of inputs $(x_0,\ x_1, x_2..., x_n)$ is termed convolution, which forms the fundamental idea behind CNN. Can be represented mathematically as $(x*w)_t$, where x represents the inputs, w represents the weights/filter/kernel, and t represents the time.
- When you convolve an image using a filter (of size, say $m\ x\ m$) with all 1's, it blurs the image. On the other hand, if you convolve an image using a filter (of size, say $m\ x\ m$) with the value at its center higher than rest of the values, it sharpens the image.
- Result of convolving an image (in general, any input) using a filter is called a feature *map*. In practice, there could be multiple filters (each serving different purposes, like blurring, sharpening, detecting edges etc) used in a CNN, each resulting in a series of *feature maps*.
- 1D filter slides over 1D input resulting in a 1D feature map.
- 2D filter slides over 2D input (horizontally, and then vertically) resulting in 2D feature map. Multiple such filters will result in multiple 2D feature maps, which can be interpreted as a 3D feature map.
- 3D filter slides over 3D input (with multiple channels occupying the *depth* of the input) resulting in 2D (not 3D) feature map. Note that this is only because, we're assuming (for sake of simplicity) that the the *depth* of the filter is the same as the *depth* of the input. Multiple such filters will result in multiple 2D feature maps, which can be interpreted as a 3D feature map.
- When a filter of size $F\ x\ F$ is applied over an input image $W_1\ x\ H_1$ that has an added padding P and also uses a stride S, it produces a feature map whose width and height is given by

$$W_2 = \frac{W_1 - F\ +\ 2P}{S}\ + 1 \qquad (25)$$
$$H_2 = \frac{H_1 - F\ + 2P}{S}\ +\ 1.$$

  Note that stride S is used to to stride horizontally, as well as vertically.
- Depth of the feature map is the given by the number of filters, typically denoted as K
- If the classification problem were to be treated as an ML problem, the image would be flattened, before applying an ML algorithm (like SVM) could be applied on it. Alternatively, we could first detect the edges of the shape using a suitable filter and hence sparsify the problem. As a third alternative, we could use SIFT/HOG algorithm to the image and transform it, before proceeding. The last two methods are called $feature\ engineering$.
- Deep learning aims to learn the weights that constitute a filter, in addition to learning the weights of the network. We could create multiple such filters, each of which could

create feature maps. It's also possible to learn multiple layers, each of which has multiple such filters.

- In CNN, the filter encapsulates weights shared across multiple *patches* of the image. In addition, the connections (between the CNN layers) are very sparse.
- In the classical feed-forward neural network, each neuron in the next layer is connected to each neuron in the previous layer, and hence results in a huge number of weights. However, in CNN, layers are divided into blocks of neurons (not necessarily contiguous) and each block in the next layer share the same weights with each block in the previous layer.
- We've multiple filters applied at each layer, so that eventually it doesn't lead to underfitting due to the above characteristics.
- Pooling operation reduces the input dimensions. Max-pooling applies a sliding filter and produces the maximum among pixels/neurons for each *patch*. Similarly, average-pooling produces the average of all pixels/neurons in each *patch*. For example, pooling using a filter size of $2\ x\ 2$ would result in $1/4^{th}$ the input dimensions.
- Applying a 2d convolution over a 3d volume keeps the depth the same, while applying 1d convolution multiple times (with multiple filters) changes the depth to the number of filters used.
- Shown below is an example of a CNN, with 6 layers, excluding the input and output layers.



- Here are the dimension calculations.
  - Input is a $32\ x\ 32\ x\ 1$ image.
  - Layer-1 (convolution) has a width and height given by the eq.(25), $\frac{32-5+0}{1}+1=28.$ Thus, the dimension of this layer's output is $28\ x\ 28\ x\ 6$, including the 6 filters.
  - Layer-2 (pooling) has a width and height given by the eq.(25),

$\frac{28-2+0}{2}+1=14$. Thus, the dimension of this layer's output is $14\ x\ 14\ x\ 6$, including the 6 filters in the previous layer.
  - Layer-3 (convolution) has a width and height given by the eq.(25), $\frac{14-5+0}{1}+1=10$. Thus, the dimension of this layer's output is $10\ x\ 10\ x\ 16$, including the 16 filters.
  - Layer-4 (pooling) has a width and height given by the eq.(25), $\frac{10-2+0}{2}+1=5$. Thus, the dimension of this layer's output is $5\ x\ 5\ x\ 16$, including the 16 filters in the previous layer.
- Here are the parameter calculations for the sparse layers.
  - Layer-1 (convolution) has 6 filters, each of which is $5\ x\ 5\ x\ 1$, thus resulting in 150 parameters.
  - Layer-2 (pooling) has no parameters, since pooling doesn't involve any transformation of its input.
  - Layer-3 (convolution) has 16 filters, each of which is $5\ x\ 5\ x\ 6$ (6 is the depth of the previous layer), thus resulting in 2400 parameters.
  - Layer-4 (pooling) has no parameters.
- Here are the parameter calculations for the fully connected layers.
  - Input to FC1 is $5\ x\ 5\ x\ 16$. FC1 has 120 neurons, each of which has biases. Thus, the number of parameters in this layer is $5*5*16*120\ +\ 120=48120$
  - Input to FC2 is 120. FC1 has 84 neurons, each of which has biases. Thus, the number of parameters in this layer is $120*84\ +84=10164$.
  - Input to Output layer is 84. FC1 has 10 neurons, each of which has biases. Thus, the number of parameters in this layer is $84*10\ +10=850$.
- Bias at the filter level adds to the number of parameters, but doesn't change the dimensions.
- Another example. https://discourse.onlinedegree.iitm.ac.in/t/sep-23-q-54-55-can-someone-explain-how-the-given-answer-is-arrived-at/141260
- Generally, in a CNN, the fully connected layers have the maximum number of parameters, as compared to the sparse layers.
- Training a CNN is similar to ANN, where only a few weights are active and rest of the weights are zero. Forward and backpropagation works similarly.
- Watch this TA session on backpropagation. Note that this technique would work only if stride $S=1$
- Here is another (detailed) example of backpropagation for CNN.

Given

$$X = \begin{bmatrix} 1 & -1 & 2 & -2 \\ 0 & 0 & 0 & 0 \\ 1 & -1 & 2 & -2 \end{bmatrix},\ y = 0.1,\ K = [0.6\ -0.2], S = 2$$

Forward propagation

Layer-1: Convolution

$$A_1 = \begin{bmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{bmatrix} = \begin{bmatrix} 0.8 & 1.6 \\ 0.8 & 1.6 \end{bmatrix}$$

where,

$$a_{00} = K_0.X_{00} + K_1.X_{01}$$
$$a_{01} = K_0.X_{02} + K_1.X_{03}$$
$$a_{10} = K_0.X_{20} + K_1.X_{21}$$
$$a_{11} = K_0.X_{22} + K_1.X_{23}$$

$\Rightarrow$

$$\frac{\partial a_{00}}{\partial K_0} = X_{00} = 1,\ \frac{\partial a_{00}}{\partial K_1} = X_{01} = -1$$
$$\frac{\partial a_{01}}{\partial K_0} = X_{02} = 2,\ \frac{\partial a_{01}}{\partial K_1} = X_{03} = -2$$
$$\frac{\partial a_{10}}{\partial K_0} = X_{20} = 1,\ \frac{\partial a_{10}}{\partial K_1} = X_{21} = -1$$
$$\frac{\partial a_{11}}{\partial K_0} = X_{22} = 2,\ \frac{\partial a_{11}}{\partial K_1} = X_{23} = -2$$

Layer-2: Sigmoid activation

$$H_1 = \begin{bmatrix} h_{00} & h_{01} \\ h_{10} & h_{11} \end{bmatrix} = \sigma\left(\begin{bmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{bmatrix}\right) = \begin{bmatrix} \sigma(0.8) & \sigma(1.6) \\ \sigma(0.8) & \sigma(1.6) \end{bmatrix} = \begin{bmatrix} 0.69 & 0.83 \\ 0.69 & 0.83 \end{bmatrix}$$

$\Rightarrow$

$$\frac{\partial h_{00}}{\partial a_{00}} = \sigma(a_{00})(1 - \sigma(a_{00})) = 0.69 * (1 - 0.69) = 0.21$$
$$\frac{\partial h_{01}}{\partial a_{01}} = \sigma(a_{01})(1 - \sigma(a_{01})) = 0.83 * (1 - 0.83) = 0.14$$
$$\frac{\partial h_{10}}{\partial a_{10}} = \sigma(a_{10})(1 - \sigma(a_{10})) = 0.69 * (1 - 0.69) = 0.21$$

$$\frac{\partial h_{11}}{\partial a_{11}} = \sigma(a_{11})(1-\sigma(a_{11})) = 0.83 * (1-0.83) = 0.14$$

Layer-3: Sum H_1

$$A_2 = h_{00} + h_{01} + h_{10} + h_{11} = 3.04$$

$\Rightarrow$

$$\frac{\partial A_2}{\partial h_{00}} = 1,\ \frac{\partial A_2}{\partial h_{01}} = 1,\ \frac{\partial A_2}{\partial h_{10}} = 1,\ \frac{\partial A_2}{\partial h_{11}} = 1$$

Layer-4: Compute $\widehat{y}$

$$\widehat{y} = \sigma(A_2) = 0.95$$

$\Rightarrow$

$$\frac{\partial \widehat{y}}{\partial A_2} = \sigma(A_2)(1-\sigma(A_2)) = 0.95 * 0.05 = 0.05$$

$$Loss = \frac{1}{2}(\widehat{y}-y)^2 = 0.5 * (0.95\ -\ 0.1) =\ 0.36$$

$\Rightarrow$

$$\frac{\partial \mathcal{L}}{\partial \widehat{y}} = \widehat{y} - y = 0.95 - 0.1 = 0.94$$

Backward propagation

Our objective is to compute $\frac{\partial \mathcal{L}}{\partial K_0}$ and $\frac{\partial \mathcal{L}}{\partial K_1}$

Going back through the layers starting with $\widehat{y}$,

$$\frac{\partial \mathcal{L}}{\partial A_2} = \frac{\partial \mathcal{L}}{\partial \widehat{y}} * \frac{\partial \widehat{y}}{\partial A_2} = 0.94 * 0.05 = 0.05$$

$$\frac{\partial \mathcal{L}}{\partial h_{00}} = \frac{\partial \mathcal{L}}{\partial A_2} * \frac{\partial A_2}{\partial h_{00}} = 0.05 * 1 = 0.05$$
$$\frac{\partial \mathcal{L}}{\partial h_{01}} = \frac{\partial \mathcal{L}}{\partial A_2} * \frac{\partial A_2}{\partial h_{01}} = 0.05 * 1 = 0.05$$
$$\frac{\partial \mathcal{L}}{\partial h_{10}} = \frac{\partial \mathcal{L}}{\partial A_2} * \frac{\partial A_2}{\partial h_{10}} = 0.05 * 1 = 0.05$$
$$\frac{\partial \mathcal{L}}{\partial h_{11}} = \frac{\partial \mathcal{L}}{\partial A_2} * \frac{\partial A_2}{\partial h_{11}} = 0.05 * 1 = 0.05$$

$$\frac{\partial \mathcal{L}}{\partial a_{00}} = \frac{\partial \mathcal{L}}{\partial h_{00}} * \frac{\partial h_{00}}{\partial a_{00}} = 0.05 * 0.21 = 0.01$$
$$\frac{\partial \mathcal{L}}{\partial a_{01}} = \frac{\partial \mathcal{L}}{\partial h_{01}} * \frac{\partial h_{01}}{\partial a_{01}} = 0.05 * 0.14 = 0.01$$
$$\frac{\partial \mathcal{L}}{\partial a_{10}} = \frac{\partial \mathcal{L}}{\partial h_{10}} * \frac{\partial h_{10}}{\partial a_{10}} = 0.05 * 0.21 = 0.01$$
$$\frac{\partial \mathcal{L}}{\partial a_{11}} = \frac{\partial \mathcal{L}}{\partial h_{11}} * \frac{\partial h_{11}}{\partial a_{11}} = 0.05 * 0.14 = 0.01$$

$$\frac{\partial \mathcal{L}}{\partial K_0} = \frac{\partial \mathcal{L}}{\partial a_{00}} * \frac{\partial a_{00}}{\partial K_0} + \frac{\partial \mathcal{L}}{\partial a_{01}} * \frac{\partial a_{01}}{\partial K_0} + \frac{\partial \mathcal{L}}{\partial a_{10}} * \frac{\partial a_{10}}{\partial K_0} + \frac{\partial \mathcal{L}}{\partial a_{11}} * \frac{\partial a_{11}}{\partial K_0} = 0.06$$

$$\frac{\partial \mathcal{L}}{\partial K_1} = \frac{\partial \mathcal{L}}{\partial a_{00}} * \frac{\partial a_{00}}{\partial K_1} + \frac{\partial \mathcal{L}}{\partial a_{01}} * \frac{\partial a_{01}}{\partial K_1} + \frac{\partial \mathcal{L}}{\partial a_{10}} * \frac{\partial a_{10}}{\partial K_1} + \frac{\partial \mathcal{L}}{\partial a_{11}} * \frac{\partial a_{11}}{\partial K_1} = -0.06$$

Now, using the update rule on K,

$$K_0 = K_0 - \eta \frac{\partial \mathcal{L}}{\partial K_0} = 0.6 - (1 * 0.06) = 0.54$$
$$K_1 = K_1 - \eta \frac{\partial \mathcal{L}}{\partial K_1} = -0.2 + (1 * 0.06) = -0.14$$

Thus, the updated filter $K = [0.54 \quad -0.14]$

Problems

1. An input image of dimension $55 \times 55 \times 3$ is to be convolved with 10 filters (kernels) of spatial dimension 5×5. In all the sub-questions, assume a bias term for kernels. Suppose that we pad zeros of size 1 (p = 1) around the sides of the image (in all channels). Assume stride = 2 for all 10 convolution filters on the input image. All the 10 filters have bias associated with it.

Suppose we flatten the output of the convolution layer above and apply a 1-D kernel of dimension$100\ x\ 1$ ($stride = 10$). Result of this convolution operation is then connected to a fully connected layer of dimension $10\ x\ 1$ with a weight matrix W and bias b. How many parameters are in the entire network?

Convolution layer

Width/Height of the convolution layer = $\frac{55 + (2 * 1) - 5}{2} + 1 = 27 (\because K = 5,\ P = 1,\ S = 2)$
Dimensions of the output of the convolution layer = $(27 * 27) * 10$.

Filter size = 5 x 5
Depth of the input layer = 3
Number of filters = 10
Number of parameters in the convolution layer = $(5 * 5 * 3 * 10) + 10 = 760$ ($\because \#biases\ = 10$)

Flattening layer

Dimensions of the output of the flattening layer = $7290\ (27 * 27 * 10)$

1D Convolution layer

Width/Height of the 1D convolution layer = $\frac{7290 - 100}{10} + 1 = 720$ ($\because K = 100,\ P = 0,\ S = 10$)

Filter size = 100 x 1
Depth of the input layer = 1.
Number of filters = 1
Number of parameters in the convolution layer = $100 * 1\ +\ 1 = 101$ ($\because \#biases\ = 1$)

Fully connected layer

Number of parameters in the fully connected layer = $720 * 10 + 10 = 7210$ ($\because \#biases\ = 10$)

Thus, total number of parameters in the entire network = $7210 + 101 + 760 = 8071$

Week9 - Word representations

- GIven a corpus, all unique words in it is called its vocabulary V.
- Words are represented using vectors, so that it can be *learned* - for purposes like sentiment analysis.
- Following techniques are generally used to measure similarity between word vectors:
  - Cosine similarity $\left(\frac{A.B}{|A||B|}\right)$
  - Euclidean distance $\left(\sqrt{(A_1 - B_1)^2 + (A_2 - B_2)^2 + ... + (A_n - B_n)^2}\right)$

Note: $A\ (A_1, A_2,..., A_n)$ and $B\ (B_1, B_2,..., B_n)$ are vectorial representations of the

words.

- The simplest way of representing a word as a vector is to use one-hot vector of size $|V|$. Following are the drawbacks of this approach.
  - size of the vector becomes unmanageably large, as the vocabulary becomes large.
  - one-hot vectors do not capture any notion of similarity between words, because euclidean distance between any two words is always $\sqrt{2}$. Likewise, the cosine similarity between any two words is always 0.
  - One-hot vectors are sparse. Dense representations are preferred over sparse representations during computations.
- Cosine similarity ranges from -1 to 1, where 1 indicates that the two vectors are perfectly similar (same direction), -1 indicates they are completely dissimilar (opposite direction), and 0 indicates they are orthogonal (90 degrees).
- Co-occurence matrix captures the number of times a term w_i in the vocabulary appears in the context of another term c_j, and is represented mathematically as $count(w_i,\ c_j)$. Context is defined as a window of k words, in which both terms w_i and cj occur.
- Rows in a co-occurence matrix are referred to as *target words*, and columns in the matrix are referred to as the *context*. Effectively, each row in the matrix is used to represent a *word* in the corpus. Note that it's not necessary that the number of rows is the same as number of columns in a co-occurence matrix, although they're typically equal.
- In a co-occurence matrix, certain words (especially the stop-words) occur very frequently and skews the matrix. As a solution, we could adopt either of the following solutions.
  - Ignore stop words and very frequent words.
  - Use a threshold t to populate the matrix as $X_{ij} = min(count(w_i, c_j),\ t)$
  - Use PMI instead of $count(w_i, c_j)$. This restricts the matrix values from blowing up.

$$PMI(w,\ c) = log\ \frac{count(w,\ c) * N}{count(c) * count(w)} \tag{26}$$

  where N is the total number of words.
- One of the issues with the PMI is when $count(w,\ c) = 0,\ PMI = -\infty$. To counter this, we use $PPMI$ instead, which ensures that all values in the matrix are positive.

$$\begin{aligned} PPMI(w,\ c) &= PMI(w,\ c) \quad if\ PMI(w,\ c) > 0 \\ &= 0 \qquad\qquad otherwise \end{aligned} \tag{27}$$

- Unfortunately, co-occurence matrices are also sparse, though not as sparse as one-hot vectors. Also, these representations could be extremely large sized ($|V|\ x\ |V|$,

where $|V|$ is the size of the vocabulary). As a solution, SVD is used to reduce dimensions of the matrix.

- SVD gives the best rank-k approximation to the original data X, and helps to decompose the PPMI matrix into 3 parts.

$$\begin{bmatrix} \uparrow & \cdots & \uparrow \\ u_1 & \cdots & u_k \\ \downarrow & \cdots & \downarrow \end{bmatrix}_{m \times k} \begin{bmatrix} \sigma_1 & & \\ & \ddots & \\ & & \sigma_k \end{bmatrix}_{k \times k} \begin{bmatrix} \leftarrow & v_1^T & \rightarrow \\ & \vdots & \\ \leftarrow & v_k^T & \rightarrow \end{bmatrix}_{k \times n}$$
$$= \sigma_1 u_1 v_1^T + \sigma_2 u_2 v_2^T + \cdots + \sigma_k u_k v_k^T$$

Based on the above equation, observe that $\sigma_1 u_1 v_1^T$ is the best rank-1 approximation for X. $\sigma_1 u_1 v_1^T + \sigma_2 u_2 v_2^T$ is the best rank-2 approximation for X. Similarly, $\sigma_1 u_1 v_1^T + \sigma_2 u_2 v_2^T + ... + \sigma_k u_k v_k^T$ gives the best rank-k approximation for X. Note that each of these components is an $m \; x \; n$ matrix.

The above decomposition of matrices can also be written concisely as follows.

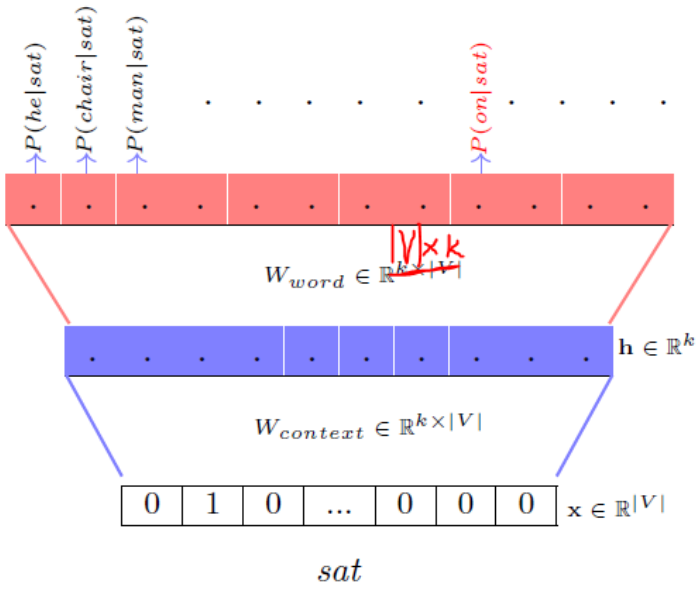
$$X = X_{PPMI(mxn)} = U_{mxk} \Sigma_{kxk} V_{kxn}^T$$

- Note that $u \in \mathbb{R}^m$, $v \in \mathbb{R}^n$, $\sigma \in \mathbb{R}^1$. Thus, rank-1 approximation will have $m + n + 1$ entries, rank-2 approximation will have $2(m + n + 1)$ entries and so on until rank-k approximation will have $k(m + n + 1)$ entries
- One very interesting outcome of SVD-based compression is that the resultant co-occurence matrix $\widehat{X}$ is dense and certain previously-hidden patterns (latent semantics) becomes visible now. Thus, certain previously-zero entries in the matrix would have non-zero entries now.
- The ij^{th} entry of XX^T captures the cosine similarity between i^{th} and j^{th} words in the vocabulary.

$$\begin{matrix} X[i:] \\ X[j:] \end{matrix} \underbrace{\begin{bmatrix} 1 & 2 & 3 \\ 2 & 1 & 0 \\ 1 & 3 & 5 \end{bmatrix}}_{X} \underbrace{\begin{bmatrix} 1 & 2 & 1 \\ 2 & 1 & 3 \\ 3 & 0 & 5 \end{bmatrix}}_{X^T}$$

- Incidentally, when we do $\widehat{X}\widehat{X}^T$ (where $\widehat{X} = SVD(X)$), the resultant matrix contains the cosine similarities between words and contexts. This can be shown to be same as $W_{word} W_{word}^T$, where $W_{word} = U\Sigma$. Thus, even after the dimensionality reduction due to SVD approximation (from $V \; x \; V$ to $V \; x \; k$), the cosine similarity between words and contexts remain unchanged.
- **Continuous bag of words**: Predict n^{th} (target) word given previous $n - 1$ (context)

words. This can be thought of as a classification problem, given a true one-hot vector representation of the n^{th} word. Typically, $n = 4$. Training data comprises of all n-word windows in the corpus. In each window, $n - 1$ (context) words are considered as the input and the n^{th} (target) word is considered as the output. THe following diagram represents a CBoW model with $n = 2$.



The input to the model is a set of context words (in this case, only one) surrounding the target word and are represented as one-hot vectors of dimension $|V|$. The hidden layer multiplies the input with a weight matrix $W_{context}$ of dimension $k\ x\ V$, thus projecting them into a lower-dimensional space of size k. The output represents the average of the projected context word vectors.

The output layer takes the k-dimensional vector from the hidden layer and multiplies it with another weight matrix W_{word} of dimensions $Vx\ k$, thus projects the hidden layer output back to the vocabulary size V, producing a vector of unnormalized scores for each word in the vocabulary. Then, this gets passed through a softmax function to obtain a probability distribution of the target word.

- We'll use the cross-entropy loss for the purpose of learning, and use backpropagation to train $W_{context}$ and W_{word}

$$h = W_{context}.x_c = u_c\ (let)$$
$$o = W_{word}.h = v_w\ (let)$$
$$\hat{y}_w = \frac{exp(u_c.v_w)}{\sum_{w' \in V} exp(u_c.v_{w'})} \tag{28}$$

Since we are using cross-entropy loss, $\mathcal{L}(\theta) = -\log \hat{y}_w$. Substituting and solving, we get

$\boxed{\nabla_{v_w} = -u_c(1-\hat{y}_w)}$. Now, use the update rule $v_w = v_w - \eta\nabla_{v_w}$

- If the model predicts target word v_w with probability score of 1 (that is, $\widehat{y_w} = 1$), then no elements in v_w or $v_{w'}$ will get modified further.
- If the model predicts target word v_w with probability score of 0.5 (that is, $\widehat{y_w} = 0.5$), then v_w will be modified as $v_w = v_w + 0.5u_c^T$ and $v_{w'}$ will be modified as $v_{w'} = v_{w'} - \widehat{y_{w'}}u_c^T$ during the next iteration.

NOTE1: w is the index of the correct output word, w' is the index of words other than the correct word, and c is the index of the context word.
NOTE2: The above examples uses a window size of 2, and $2-1=1$ context word; If the window size is n, there'll be $n-1$ context words and hence $h = \sum_{i=1}^{d-1} u_{c_i}$.

- Essentially, the training objective ensures that cosine similarity between word (v_w) and context word (u_c) is maximized. This will transitively also pull certain words close to each other- say v_w and $v_{w'}$.
- Skip – gram (Word2Vec) model helps with predicting context words given the input (target) word. The loss function used is the sum of cross-entropy of all predictions, $\mathcal{L}(\theta) = -\sum_{i=1}^{d-1} log\,\widehat{y_{w_i}}$, where d is the number of context words being predicted. In comparison to the CBoW model, the weights are interchanged between the layers - the first layer uses W_{word} and the second layer uses $W_{context}$ matrices.
- Skip-gram is slower compared to CBoW, since the former needs to predict multiple context words, whereas the the latter needs to predict only one target word. Softmax function at the output is computationally expensive, and there exists 3 solutions to that - negative sampling, contrastive estimating and hierarchical softmax.
- In negative sampling, two sets of pairs are chosen - $(w,\ c)$ picked from set D of all valid pairs, and $(w,\ r)$ picked from set D' of all invalid pairs. Note that c is a context word, and r is a non-context word picked randomly from a $modified\ unigram$ distribution $r \sim p(r)^{\frac{3}{4}}$, where $p = \frac{count(r)}{|V|}$. Note that V is the vocabulary. In the above formula, exponent $\frac{3}{4}$ is tuned using a hyper-parameter.
- Note that the size of D' is k times that of D, where k is a hyper-parameter and has a positive value. The right value of k is arrived at through tuning by experimenting with different validation sets.
- For every $(w,\ c)$ pair, we want to maximize the probability $p(z=1|w,c)$ that the

selected pair is valid. It can be modelled as the sigmoid of the dot product of u_c and v_w and can be mathematically represented as

$p(z = 1|w, c) = \frac{1}{1 + e^{-u_c^T v_w}} = \sigma\left(u_c^T v_w\right)$. Hence the objective function is

$$\underset{\theta}{maximize} \prod_{(w,c)\in D} p(z = 1|w, c)$$

- Similarly, for every (w, r) pair, we want to maximize the probability $p(z = 0|w, r)$ that the selected pair is invalid. It can be modelled in a similar manner and can be mathematically represented as

$p(z = 0|w, r) = 1 - \frac{1}{1 + e^{-u_r^T v_w}} = \frac{1}{1 + e^{u_r^T v_w}} = \sigma\left(-u_r^T v_w\right)$. Hence the objective function is $\underset{\theta}{maximize} \prod_{(w,r)\in D'} p(z = 0|w, r)$

- Combining the above two cases, the final objective function is

$$\underset{\theta}{maximize} \prod_{(w,c)\in D} p(z = 1|w, c) \prod_{(w,r)\in D'} p(z = 0|w, r)$$

- Taking log of the above formula, the objective function can be rewritten as

$$\underset{\theta}{maximize} \sum_{(w,c)\in D} \log \sigma(v_c^T v_w) + \sum_{(w,r)\in D'} \log \sigma(-v_r^T v_w)$$

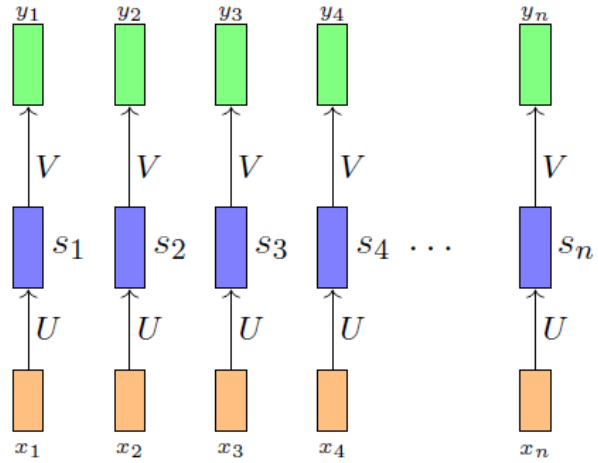
- Thus, the skip-gram model ensures that the word and context of a valid pair are brought together, whereas the word and context of an invalid pair and separated from each other.
- Similar to the exponent used in the case of negative sampling, when hyper-parameters are introduced into SVD based model, it starts performing equal or even better than word2vec model.
- In contrastive estimating, we predict a single score s_r instead of softmax probabilities on all context words, indicated by s. In the output layer, the weight matrix is of size $1\ x\ k$. We need $(s - s_r) \geqslant m$, where m is a large positive value. Thus, the loss function is $\mathcal{L}(\theta) = s - (s_r + m)$. Objective function is $maximize\ max(0,\ s - (s_r + m))$

Week10 - RNN

- RNN deals with dynamic inputs, unlike CNN and ANNs that deal with fixed size inputs

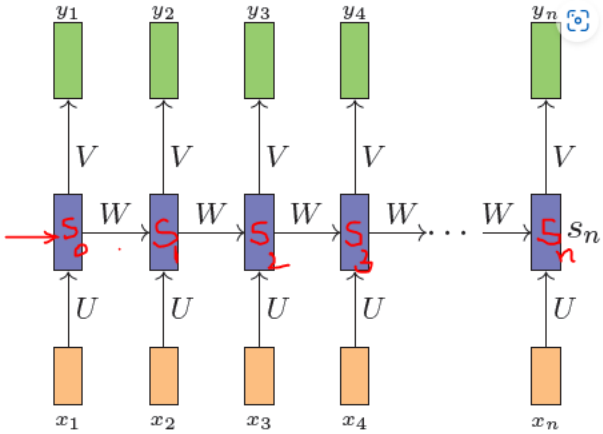
and has no inter-dependence between its inputs.

- When the model must learn and produce outputs that depend on the current input and previous outputs, we're dealing with sequence learning problems. These models can be thought of learning and producing outputs at every $time\ step$.
- There are some problems (like sentiment analysis) that require to produce an output only during the last time step, instead of each time step.
- Given below is a model that runs through n time steps (left to right)



At the first time-step, input is x_1, at the second time-step, input is x_2 and at the n^{th} time step, input is x_n. Note that at each time-step model remains *unchanged* - input x_i is multiplied by U and fed into s_i, output of which is multiplied by V and produce y_i.

However the time steps remain independent of each other, until we feed the previous *hidden* state s_{i-1} to the current *hidden* state s_i as follows.



This network (called RNN) can be mathematically represented as follows:

$$s_i = \sigma(Ws_{i-1} + Ux_i + b)$$
$$y_i = softmax(Vs_i + c)$$

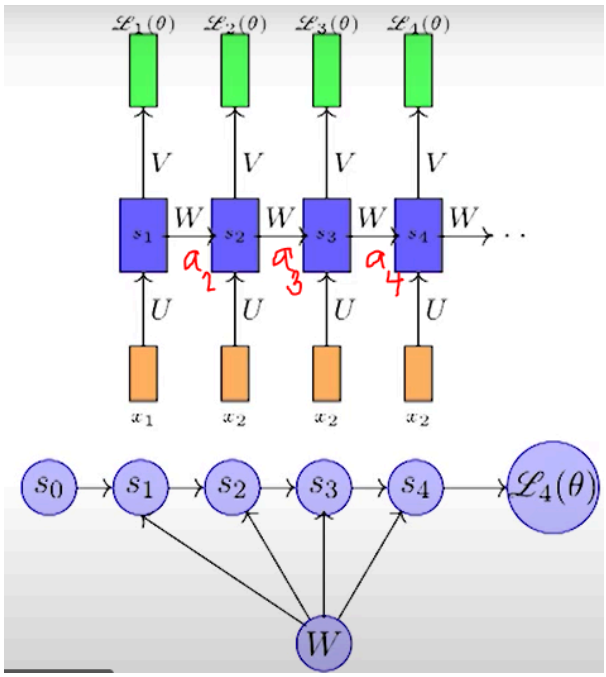
Note that σ is a non-linear function (typically *tanh*) used at each time-step. Also, note that b and c are biases in each time-step.

- If $x_i \in \mathbb{R}^n$, $s_i \in \mathbb{R}^d$, $y_i \in \mathbb{R}^k$ (say, k classes), then $U \in \mathbb{R}^{dxn}$, $V \in \mathbb{R}^{kxd}$ and $W \in \mathbb{R}^{dxd}$
- The network is trained using backpropagation with time (BPTT). It'll use cross-entropy loss at each timestep and softmax function to compute each output. Thus, total loss

$$\mathcal{L}(\theta) = \sum_{t=1}^{T} -log(y_t c))$$

NOTE: In the above formula, t denotes the timestep and c denotes the true class.

- Before we can start the calculations on backpropagation of the loss, let's redraw the above diagram as follows.



- Computing the derivative with respect to V is straight-forward. $\frac{\partial \mathcal{L}(\theta)}{\partial V} = \sum_{t=1}^{T} \frac{\partial \mathcal{L}_t(\theta)}{\partial V}$
- Howevver, computing the derivative with respect to W is not so. $\mathcal{L}_4(\theta)$ depends on s_4 & W; s_4 in turn depends on s_3 & W and so on until s_0. Hence, although $s_4 = \sigma(Ws_3 + b)$, we can't compute $\frac{\partial s_4}{\partial W}$ by considering s_3 as a constant.

$$\frac{\partial \mathcal{L}(\theta)}{\partial W} = \sum_{t=1}^{T} \frac{\partial \mathcal{L}_t(\theta)}{\partial W} = \sum_{t=1}^{T} \frac{\partial \mathcal{L}_t(\theta)}{\partial s_t} * \frac{\partial s_t}{\partial W} = \sum_{t=1}^{T} \left(\frac{\partial \mathcal{L}_t(\theta)}{\partial s_t} * \sum_{k=1}^{t} \frac{\partial s_t}{\partial s_k} \frac{\partial^+ s_k}{\partial W} \right)$$

- Following steps prove that in the case of RNN, gradients could explode or vanish.

In the equation above for $\frac{\partial \mathcal{L}(\theta)}{\partial W}$, we can write $\frac{\partial s_t}{\partial s_k}$ as $\frac{\partial s_t}{\partial s_{t-1}} \frac{\partial s_{t-1}}{\partial s_{t-2}} .. \frac{\partial s_{k+1}}{\partial s_k}$

$$= \prod_{j=k}^{t-1} \frac{\partial s_{j+1}}{\partial s_j} = \prod_{j=k+1}^{t} \frac{\partial s_j}{\partial s_{j-1}}$$

But, $\frac{\partial s_j}{\partial s_{j-1}} = \frac{\partial s_j}{\partial a_j} \frac{\partial a_j}{\partial s_{j-1}}$.

Since $a_j = [a_{j1}, a_{j2}, a_{j3} ... a_{jd}]$ and $s_j = [\sigma(a_{j1}), \sigma(a_{j2}), \sigma(a_{j3}) ... \sigma(a_{jd})]$

$\frac{\partial s_j}{\partial a_j}$ is a diagonal matrix $\begin{pmatrix} \sigma'(a_{j1}) & 0 & 0 & 0 \\ 0 & \sigma'(a_{j2}) & 0 & 0 \\ 0 & 0 & \ddots & 0 \\ 0 & 0 & \dots & \sigma'(a_{jd}) \end{pmatrix} = diag(\sigma'(a_j))$

Since $a_j = Ws_{j-1} + b$ and $s_j = \sigma(a_j)$, $\frac{\partial a_j}{\partial s_{j-1}} = W^T$

$$\frac{\partial s_j}{\partial s_{j-1}} = diag(\sigma'(a_j)).\, W$$

$$\implies \left|\left| \frac{\partial s_j}{\partial s_{j-1}} \right|\right| = \left|\left| diag(\sigma'(a_j)).\, W \right|\right|$$

$$\implies \left|\left| \frac{\partial s_j}{\partial s_{j-1}} \right|\right| \leqslant \left|\left| diag(\sigma'(a_j)) \right|\right| . \left|\left| W \right|\right|$$

Note that $\sigma'(a_j) \leqslant 0.25$ if σ is sigmoid.. Also, note that $\sigma'(a_j) \leqslant 1$ if σ is tanh.

Hence, $\left|\left| \frac{\partial s_j}{\partial s_{j-1}} \right|\right| \leqslant \gamma\lambda$, where γ represents the bound.

This implies that $\frac{\partial s_t}{\partial s_k} \leqslant (\gamma\lambda)^{t-k}$, which in turn implies that it vanishes if $\gamma\lambda < 1$ and explodes if $\gamma\lambda > 1$.

- One way to counter this is not to backpropagate until s_0. Or, it's possible to clip the gradient.
- As time steps get added, the old information could get completely morphed by the later inputs. Similarly, it's difficult to attribute loss due to a much earlier time step, due to *vanishing* gradients.

- Thus, training RNNs requires careful initialization of weights and biases to ensure stable convergence during training, as poorly initialized weights can lead to gradient saturation and slow learning.
- Here's our wishlist for a $modified$ RNN used for sentiment analysis. Under each item in the wishlist, corresponding operations of a $modified$ RNN (termed $LSTM$) also have been mentioned.
  - Selectively write new information from the current word to the state.
    - *Selective write* operation could choose to write certain elements of s_{t-1} to s_t, or choose to write a certain fraction of each of the elements of s_{t-1} to s_t. This is achieved by element-wise multiplication with an $output\ gate$, whose elements lie between 0 and 1.
    - Output gate is computed as $o_{t-1} = \sigma(W_o h_{t-2} + U_o x_{t-1} + b_o)$. Output of the above *selective write* operation is given as $h_{t-1} = o_{t-1} \odot \sigma(s_{t-1})$
      NOTE: In RNN, $s_t = \sigma(Ws_{t-1} + Ux_t)$ rewrites the state at s_{t-1}.
  - Selectively read information added by previous sentiment bearing words.
    - This is done in two steps. We'll first calculate an intermediate state $\widetilde{s_t}$ using h_{t-1} (along with x_t) as $\widetilde{s_t} = Wh_{t-1} + Ux_t + b$.
    - *Selective read* operation could choose to read certain elements of $\widetilde{s_t}$. This is achieved by element-wise multiplication with an $input\ gate$, whose elements lie between 0 and 1.
    - Input gate is computed as $i_t = \sigma(W_i h_{t-1} + U_i x_t + b_i)$. Output of the above *selective read* operation is given as $i_t \odot \widetilde{s_t}$.
  - Should be able to forget the information added by stop-words.
    - Note that this step might seem redundant when *selective write* already in place. But, existence of a *selective forget* operation adds to more flexibility in the process. Again, note that these two seemingly redundant operations are combined in GRU (Gated Recurrent Units).
    - *Selective forget* operation allows to forget some fraction of the s_{t-1} vector. This is achieved by element-wise multiplication with an $forget\ gate$, whose elements lie between 0 and 1.
    - Forget gate is computed as $f_t = \sigma(W_f h_{t-1} + U_f x_t + b_f)$. Output of the above *selective read* operation is given as $f_t \odot s_{t-1}$.
    - Note that in the absence of the forget operation, $s_t = s_{t-1} + i_t \odot \widetilde{s_t}$. But, with the *selective forget*, $s_t = f_t \odot s_{t-1} + i_t \odot \widetilde{s_t}$
  - Though these operations occur in a recurrent fashion, operations in a single timestep could be thought as occurring in the order *selective read*, *selective forget* and *selective write*. Output of the LSTM module is h_t.

- The final set of equations of LSTM are as follows.

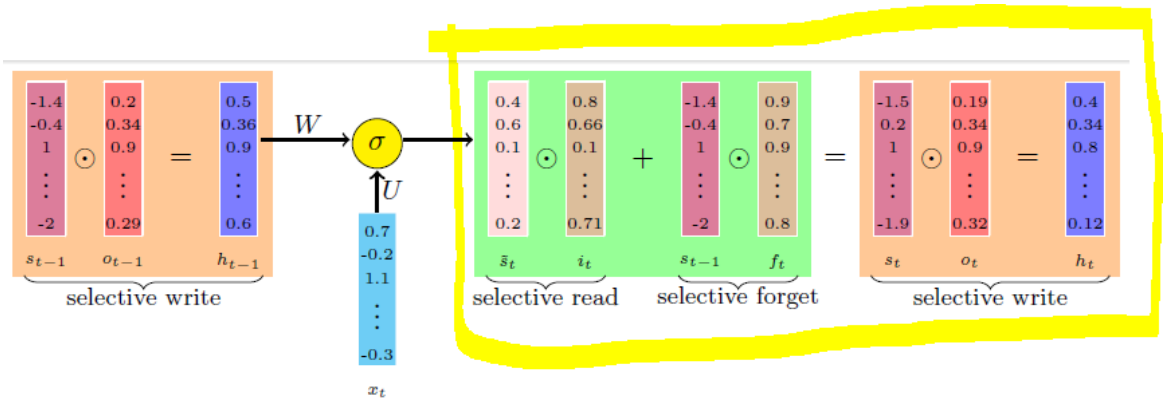
Gates:

$$o_t = \sigma(W_o h_{t-1} + U_o x_t + b_o)$$
$$i_t = \sigma(W_i h_{t-1} + U_i x_t + b_i)$$
$$f_t = \sigma(W_f h_{t-1} + U_f x_t + b_f)$$

States:

$$\tilde{s_t} = \sigma(W h_{t-1} + U x_t + b)$$
$$s_t = f_t \odot s_{t-1} + i_t \odot \tilde{s_t}$$
$$h_t = o_t \odot \sigma(s_t)$$

- All of the above operations and states is pictorially represented as follows. The highlighted part represents the set of operations that happen in one timestep.



- In a GRU, the forget gate is absent. $(1 - i_t)$ is used instead of f_t. Also, instead of using h_{t-1}, all equations will use s_{t-1}. Hence, the equations for a GRU are as follows.

Gates:

$$o_t = \sigma(W_o s_{t-1} + U_o x_t + b_o)$$
$$i_t = \sigma(W_i s_{t-1} + U_i x_t + b_i)$$

States:

$$\tilde{s_t} = \sigma(W(o_t \odot s_{t-1}) + U x_t + b)$$
$$s_t = (1 - i_t) \odot s_{t-1} + i_t \odot \tilde{s_t}$$

Problems

Given a scenario where there are 8 possible letters represented as one-hot encoded vectors of length 8, the RNN employs the following formulas for the state vector and output at time step t:

$$s_t = \sigma(Ux_i + Ws_{t-1} + b)$$
$$\hat{y}_t = O(Vs_t + c)$$

Here, σ and O denote the sigmoid and softmax functions, respectively. Assume that $s_t \in \mathbb{R}^2$ and $y_t \in \mathbb{R}^8$

- Q.1. With a total of 20 time steps (T = 20, implying prediction for a word of length 19), what is the total count of parameters (including bias) within the network?
  Ans: Since s_t is 2 x 1, each part of the first equation must be of the same shape. Thus, Ux_i must be 2 x 1. Since x_i is 8 x 1, U must be 2 x 8. Similarly, Ws_{t-1} must be 2 x 1. Since s_{t-1} is 2 x 1, W must be 2 x 2. b also must be 2 x 1 by same argument.

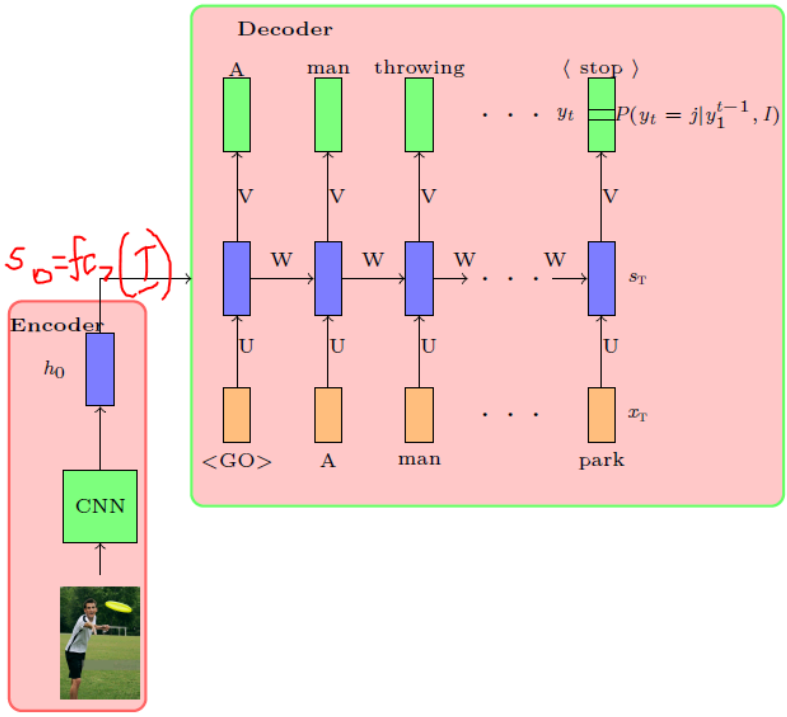
  Since y_t is 8 x 1 (and so is $\widehat{y_t}$), each part of the second equation must be of same shape. Thus, Vs_t must be 8 x 1. Since s_t is 2 x 1, V must be 8 x 2. Similarly, c must be 8 x 1.

  Total number of parameters is the sum total of all these, and hence 46.
- Q.2. If all parameters in the network are initialized to 0, what will be the $\widehat{y}$ at timestep 2?
  Ans: From the first equation, s_1 and s_2 are $\sigma(0) = [0.5,\ 0.5]$.
  $\widehat{y_t} = softmax(0_{8x1}) = \left[\frac{1}{8}, \frac{1}{8}, \frac{1}{8}, \frac{1}{8}, \frac{1}{8}, \frac{1}{8}, \frac{1}{8}, \frac{1}{8}\right]$
- Q.3. If all the parameters (including bias) in the network are initialized to zero, what will be the total loss after 20 time steps (assume that indices start with 1) for the input $[0,\ 0,\ 0,\ 1,\ 0,\ 0,\ 0,\ 0]^T$? Assume the loss to be cross-entropy at each time step.
  Ans: At each timestep, the loss is $-ln(\widehat{y}) = -ln\left(\frac{1}{8}\right) = 2.08$. Since there are 19 timesteps (not 20) to final output, the total loss = $2.08 * 19 = 39.5$

Week11 - Encoder-Decoder Models & Attention

- RNN can be represented compactly as $s_t = RNN(s_{t-1}, x_t)$. We model this as $P(y_t = j|y_1, y_2, y_3 \ldots y_{t-1})$, which is often represented as $P\left(y_t = j|y_1^{t-1}\right)$ in shorthand notation. Here j is the word embedding from vocabulary, that has been assigned the maximum probability at time step $t-1$. Alternatively, we can write this as $P(y_t = j|s_t) = softmax(Vs_t + c)_j$, since s_t is surrogate for the previous y_{t-1}.

- GRU can be represented compactly as $s_t = GRU(s_{t-1}, x_t)$
- LSTM can be represented compactly as $s_t = LSTM(h_{t-1}, s_{t-1}, x_t)$
- If we're trying to write a caption for a given image I, we can formally represent the task as $P(y_t|s_t,\ I)$. Since I is the result of a fully connected layer in a CNN (encoder), we can rewrite this as $P(y_t|s_t,\ fc_7(I))$



- Note that the encoder block reads inputs only once and encodes it to create an *embedding*. The decoder block uses this as its *zero state*(s_0) and produces an output $\widehat{y_i}$ for every input x_i. Alternatively, use the *embedding* produced by the encoder block at every timestep.
- Applications of the Encoder-decoder model architecture.
  - For image captioning, the model is given by
    - **Encoder:**

$$s_0 = CNN(x_i)$$

    - **Decoder:**

$$s_t = RNN(s_{t-1}, e(\hat{y}_{t-1}))$$
$$P(y_t|y_1^{t-1}, I) = softmax(Vs_t + b)$$

  - For textual entailment, machine translation, transliteration, document

summarization, dialog generation the model is given by

- **Encoder:**

$$h_t = RNN(h_{t-1}, x_{it})$$

- **Decoder:**

$$s_0 = h_T \quad (T \text{ is length of input})$$
$$s_t = RNN(s_{t-1}, e(\hat{y}_{t-1}))$$
$$P(y_t|y_1^{t-1}, x) = softmax(Vs_t + b)$$

– For image question answering, the model is given by

- **Encoder:**

$$\hat{h}_I = CNN(I), \ \tilde{h}_t = RNN(\tilde{h}_{t-1}, q_{it})$$
$$s = [\tilde{h}_T; \hat{h}_I]$$

- **Decoder:**

$$P(y|q, I) = softmax(Vs + b)$$

NOTE: q_i represents the i^{th} question and used instead of x_i. The t^{th} timestep of q_i is represented as q_{it}.

– For video captioning, the model is given by

- **Encoder:**

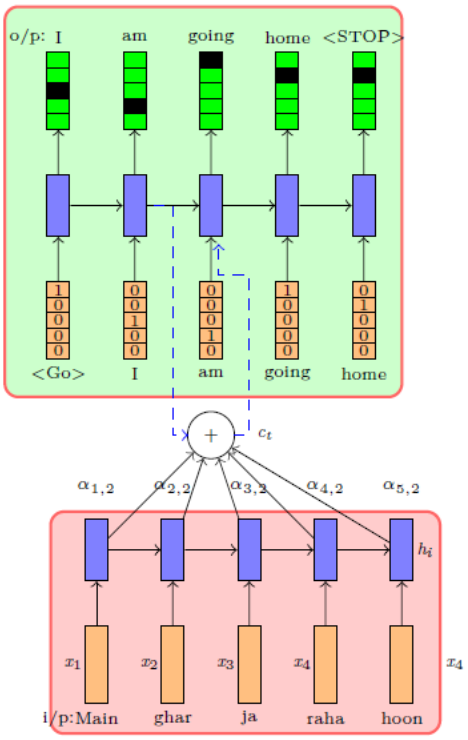
$$h_t = RNN(h_{t-1}, CNN(x_{it}))$$

- **Decoder:**

$$s_0 = h_T$$
$$s_t = RNN(s_{t-1}, e(\hat{y}_{t-1}))$$
$$P(y_t|y_1^{t-1}, x) = softmax(Vs_t + b)$$

- Encoder-decoder architecture suffers from the disadvantage of having to remember every word of the input sequence, to produce the output sequence. Instead, focus on certain words et each timestep and hence feed only the relevant words to the decoder. This is called attention mechanism and is a better modelling choice.
- The attention mechanism in RNN based encoder-decoder architecture helps the decoder to understand the context of words in a given sentence
- If multiple words in the input sequence need to be passed to the decoder, pass a weighted sum of all such words, so that the dimensions of s_0 remains unchanged. In the following figure, all words in the input sequence are *encoded,* and $attention$

weights applied to each in order to produce a single embedding. For example, weights $\alpha_{12}, \alpha_{22}, \alpha_{32}, \alpha_{42}, \alpha_{52}$ are applied to the input sequence to produce a (new) embedding , which is in turn used to *decode* the *second* word/timestep in the decoder. Note that these weights need be learnt during the training phase.



- $e_{jt} = f_{ATT}(s_{t-1}, h_j)$ captures the importance of the j^{th} input word to produce the t^{th} output word. One of the many possible choices for f_{ATT} is

$$e_{jt} = V_{att}^{T} \tanh(U_{att} s_{t-1} + W_{att} h_j)$$

where

$V_{att} \in \mathbb{R}^d$, $U_{att} \in \mathbb{R}^{d \times d}$, $W_{att} \in \mathbb{R}^{d \times d}$ are additional parameters of the model

The above formula involve three vectors (s, h, V), two linear transformations (U, W), one non-linearity ($tanh$) and a dot product (V . $tanh()$)

- Now, e_{jt} across all input words should add to 1. Attention weights are calculated using the following formula.

$$\alpha_{jt} = \frac{exp(e_{jt})}{\sum_{j=1}^{M} exp(e_{jt})}$$

also represented as,

$$\alpha_{ti} = align(y_t, h_i)$$
$$= \frac{\exp(\text{score}(\boldsymbol{s}_{t-1}, \boldsymbol{h}_i))}{\sum_{i'=1}^{n} \exp(\text{score}(\boldsymbol{s}_{t-1}, \boldsymbol{h}_{i'}))}$$

- With the introduction of attention, the model for machine translation is given by
  - **Encoder:**
    $$h_t = RNN(h_{t-1}, x_t)$$
    $$s_0 = h_T$$
  - **Decoder:**
    $$e_{jt} = V_{attn}^T tanh(U_{attn} h_j + W_{attn} s_t)$$
    $$\alpha_{jt} = softmax(e_{jt})$$
    $$c_t = \sum_{j=1}^{T} \alpha_{jt} h_j$$
    $$s_t = RNN(s_{t-1}, [e(\hat{y}_{t-1}), c_t])$$
    $$P(y_t) = softmax(Vs_t + b)$$
    $$P(y_t | y_1^{t-1}, x)$$

NOTE: c_t is the called the context vector (also called thought vector)

- Now, let's try to fix the sequential computation of the hidden states in the encoder and parallelize the computations instead. It's possible to compute the α weights for a given timestep parallely, so as to improve computing effciency. Unfortunately, the weights can't be computed for all timesteps together, since the weights depend on s_{t-1}.

Problems

Consider a simple translation task using an encoder-decoder architecture with attention:
- The encoder processes an input English sentence with 3 words: "I am happy".
- The decoder generates a French sentence with 2 words: "$J'suisheureux$".
- The hidden state size of both the encoder and decoder RNNs is 2.
- Assume we have precomputed alignment scores e_{ij} for the attention mechanism.

Given:

- Encoder hidden states $h_1 = (0.1, 0.3), h_2 = (0.4, 0.5), (h_3 = (0.6, 0.9)$
- Alignment scores for the first decoder step: $e_{11} = 0.5, e_{12} = 1.5, e_{13} = 2.5$

Find the first context word from the above information.

Answer: From the given alignment scores, we can compute the attention weights as follows:

$$\alpha_{11} = \frac{e^{0.5}}{e^{0.5} + e^{1.5} + e^{2.5}} = 0.09, \alpha_{12} = \frac{e^{1.5}}{e^{0.5} + e^{1.5} + e^{2.5}} = 0.24, \alpha_{13} = \frac{e^{2.5}}{e^{0.5} + e^{1.5} + e^{2.5}} = 0.66$$

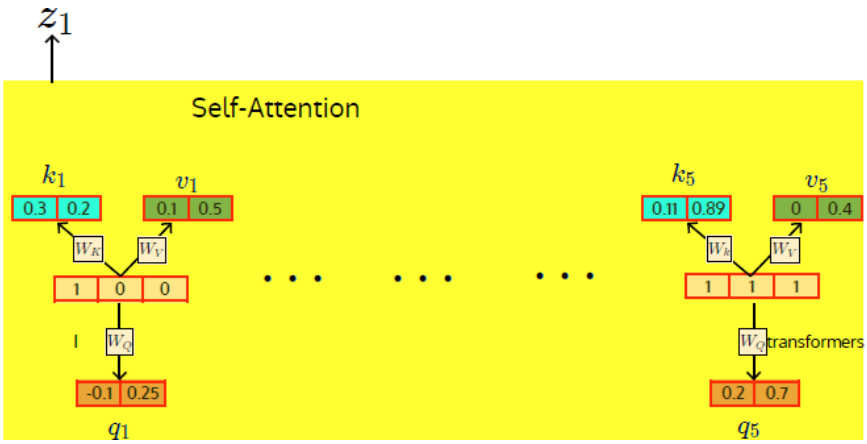
NOTE: These scores (and hence the weights) are only for the first input word.

Now, the first context word is calculated using the formula $c_t = \sum_{j=1}^{3} \alpha_{jt} h_t$ as given in one of the above sections. Thus,

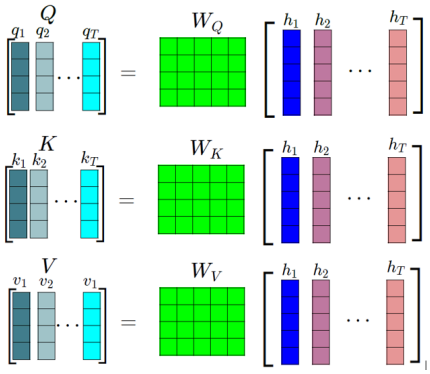
$$c_1 = \alpha_{11} h_1 + \alpha_{12} h_2 + \alpha_{13} h_3 = 0.09 * (0.1, 0.3) + 0.24 * (0.4, 0.5) + 0.66 * (0.6, 0.9) = (0.50, 0.74)$$

Week12 - Transformers

- The attention mechanism in encoder-decoder model requires the hidden states from both encoder and decoder, as evident in the equation $e_{jt} = f_{ATT}(s_{t-1}, h_j)$. However, in the case of transformer architecture, it is replaced by a self-attention block, which calculates alignment scores as $e_{jt} = f_{ATT}(h_i, h_j)$. Note that these set of computations is strictly among the words within the encoder block. Rest of the computation steps remain the same as with the encoder-decoder model.
- Very importantly, we could parallelize these set of computations, whereas in the previous case of encoder-decoder model, we couldn't do that.
- In the encoder-decoder model, $f_{ATT} = V_{ATT}^T \tanh(U_{ATT} s_{t-1} + W_{ATT} h_j)$. But, in the case of self-attention block, we will use a different formula for f_{ATT} using the vectors $h_i,\ h_j$. We'll use 3 linear transformations - termed as key, query and value - on each of these two vectors.
- $q_j = W_Q h_j, v_j = W_V h_j, k_j = W_K h_j$. Note that size of h_j is $\mathcal{R}^d$, and each of $W_K,\ W_Q,\ W_V$ is $\mathcal{R}^{dxd}$, and the resultant vectors are of size $\mathcal{R}^d$. However, it's also possible to have resultant vectors $q_j,\ v_j,\ k_j$ with different dimensions, say, $\mathcal{R}^{d_q},\ \mathcal{R}^{d_v},\ \mathcal{R}^{d_k}$ respectively, in which case the matrices will have corresponding dimensions.



- The alignment scores are given by $score(q_i,\ k_j)$, where j varies from $1\ to\ T$. This score represents how much the i^{th} output position should attend to the j^{th} input position when computing the output at position i. For example, the alignment score of the 1st output position is calcul ated as $score(q_1,\ k_j)$, where j varies from $1\ to\ T$.
- The $score$ function is typically the dot product of vectors. Thus, $e_1 = [q_1.k_1, q_1.k_1 \ldots q_1.k_5]$. In order to calculate the attention weight α_{1j}, use a softmax similar to the encoder-decoder block.
- Now, the first context word is calculated as $z_1 = \sum_{j=1}^{T} \alpha_{1j} v_j$. SImilarly, for rest of the context words.
- The above computations can be vectorized and hence parallelized. $Q = W_Q H,\ K = W_K H,\ V = W_V H$. In these equations, H is $d\ x\ T$, since it contains the h_j vectors arranged along its columns. As mentioned previously, $W_K,\ W_Q,\ W_V$ is $d\ x\ d$. Hence, $Q, K\ and\ V$ have size $d\ x\ T$.



Now, output $Z = [z_1,\ z_2 \ldots z_T] = softmax\left(\frac{Q^T K}{\sqrt{d_k}}\right) V^T$. Note that Z has size $T\ x\ d$.

- Use multiple heads, if multiple different contexts must be learnt from the same input

sequence. THis constitutes one layer in the encoder stack. Create as many layers as the architecture requires. Calculations across layers cannot be parallelized, since the layers are dependent on each other and hence need to be sequential.

- The output Z of the encoder is the input for the decoder. Decoder will have multiple layers. The first layer takes Z as input, along with the words decoded so far.