ST: 0.1a+0.4F+0.25q1+0.25q2

26 + 0.1x + 0.4 F

F = 35.0

SE: 0.05a+0.2q2+0.4f+0.1gp1+0.1gp2+0.1pp+0.05cp

38.8 + 0.05a + 0.4F + 0.05cp

F = 3.0

DL: 0.1a+Max((0.4F + 0.25Qz1 + 0.25Qz2), 0.5F+ 0.3 Max(Qz1,Qz2))

2+Max((0.4F + 27.5), (0.5F + 18))

F = 26.25 or 40

C: 0.05GAA (objective) + 0.1GAAP + 0.15Qz1 + 0.20 OPPE1 + 0.20 OPPE2 + 0.30F

53.5 + 0.05GAA (objective) + 0.1GAAP + 0.30F +(5)

F = 0.0 :)

Deep Learning

Week 1

History of deep learning:

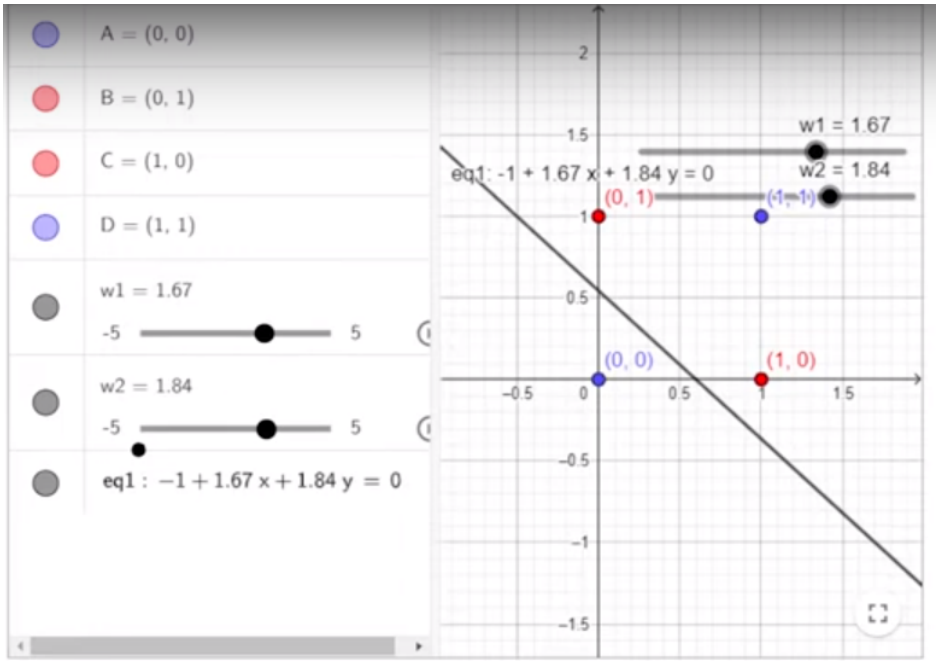
Deep learning is inspired by the biology of the brain i.e. the network of neurons. This is translated to perceptrons in computer science.

Jordan network was a RNN that introduced context to NLP and video processing.

:)

Week 2

Linearly separable is essentially when one straight line can't divide the solution plane into the required sets.



See above for XOR. This is to say that one perceptron can only handle a linearly separable data. We have 2^{2^n} boolean functions for n inputs.

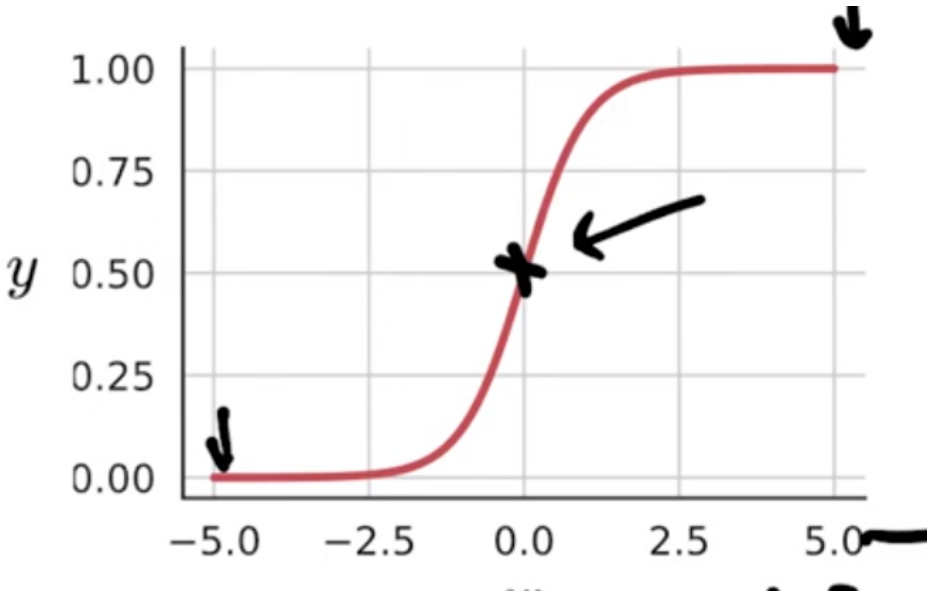
A few terms to remember:

- w is weight. This applied to an input.
- h is usually the output from a perceptron.
- b or $bias$ is usually added in the end , just before evaluation of a function.

So formula is usually: $w_0 * i_0 + ... + b < condition >< some\ value >$ so for example: $w_0 * i_0 + ... + b \geq 0$ If that is true then we have some output else some other output.

Input layer is not a perceptron. Hidden and output layers are perceptrons.

Things like sigmoid function help with non boolean implementations. So there are smooth transition between True and False.



This is usually applied to the final output.

Full formula here is

$output = \frac{1}{1+r^{-(w_1 x+b_1)}}$

Range of derivative of sigmoid is 0.25 to 0

Error:

Error is used to guide the algorithm.

Taylor series is a way of approximating a continuously differential function.

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

The higher the order of the function we consider the higher the "accuracy" will be.

$$\mathcal{L}(\theta + \eta u) = \mathcal{L}(\theta) + \eta * u^T \nabla_\theta \mathcal{L}(\theta) + \frac{\eta^2}{2!} * u^T \nabla_\theta^2 \mathcal{L}(\theta) u + \frac{\eta^3}{3!} \cdots + \frac{\eta^4}{4!} \cdot$$

This is gradient descent formula

Gradient is the collection of the partial derivatives. Matrix thingy. In gradient descent we move opposite to the gradient because that is where the loss reduces the most.

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

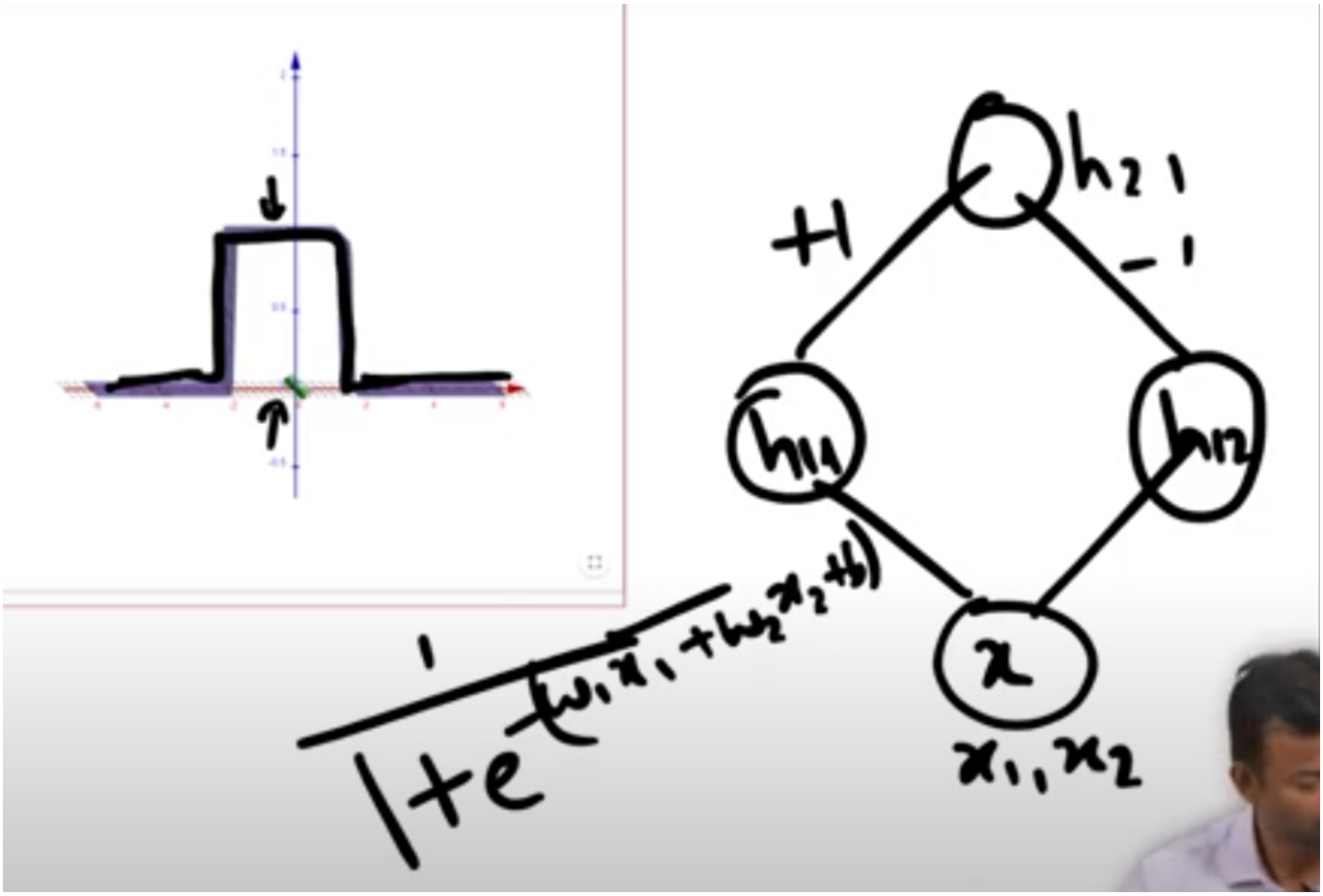
Let's say this is loss function. $f(x)$ is the prediction.

Change in weights is partial derivation (w and then b) of this.

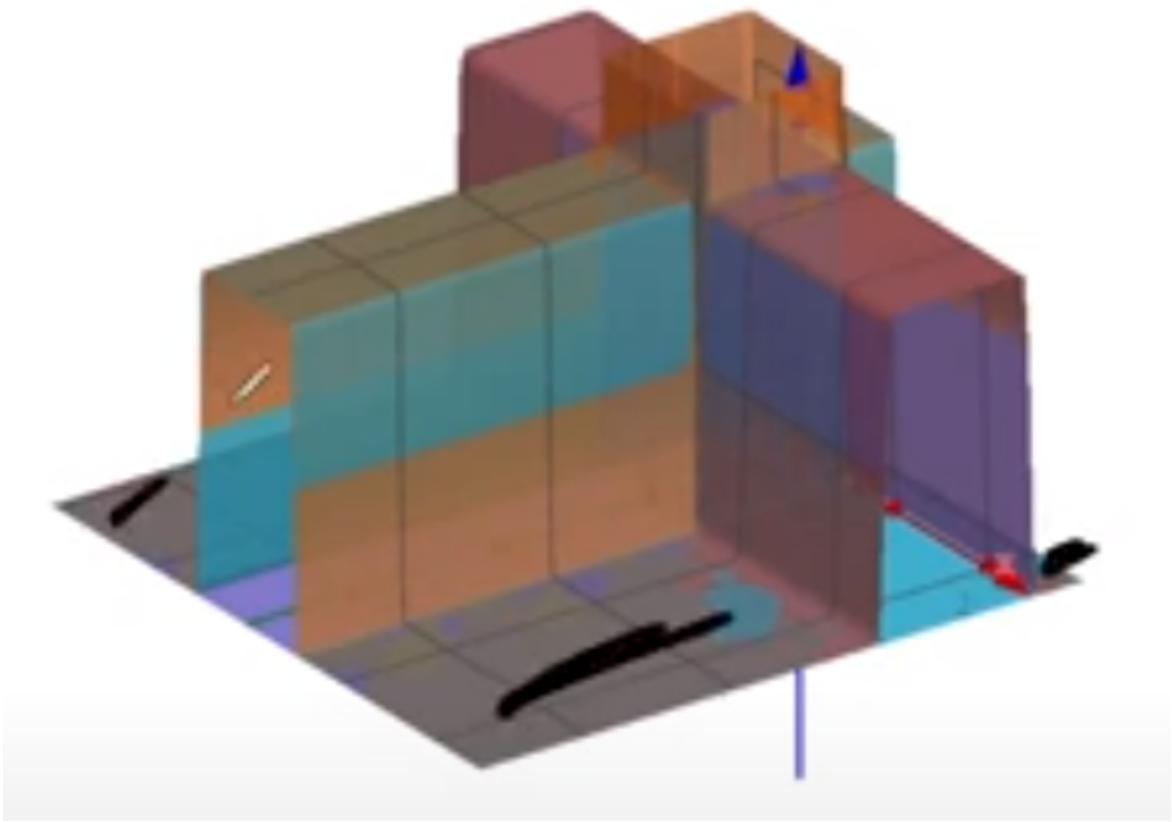
$w_1 = w_0 - \eta(\frac{df}{dw})$

REVISE THE BASIC INTEGRATION AND DERIVATION RULES!!!

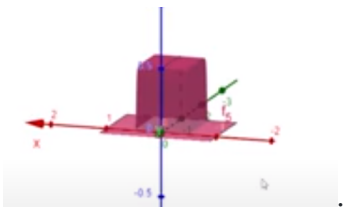
We can approximate any function with x number of towers.

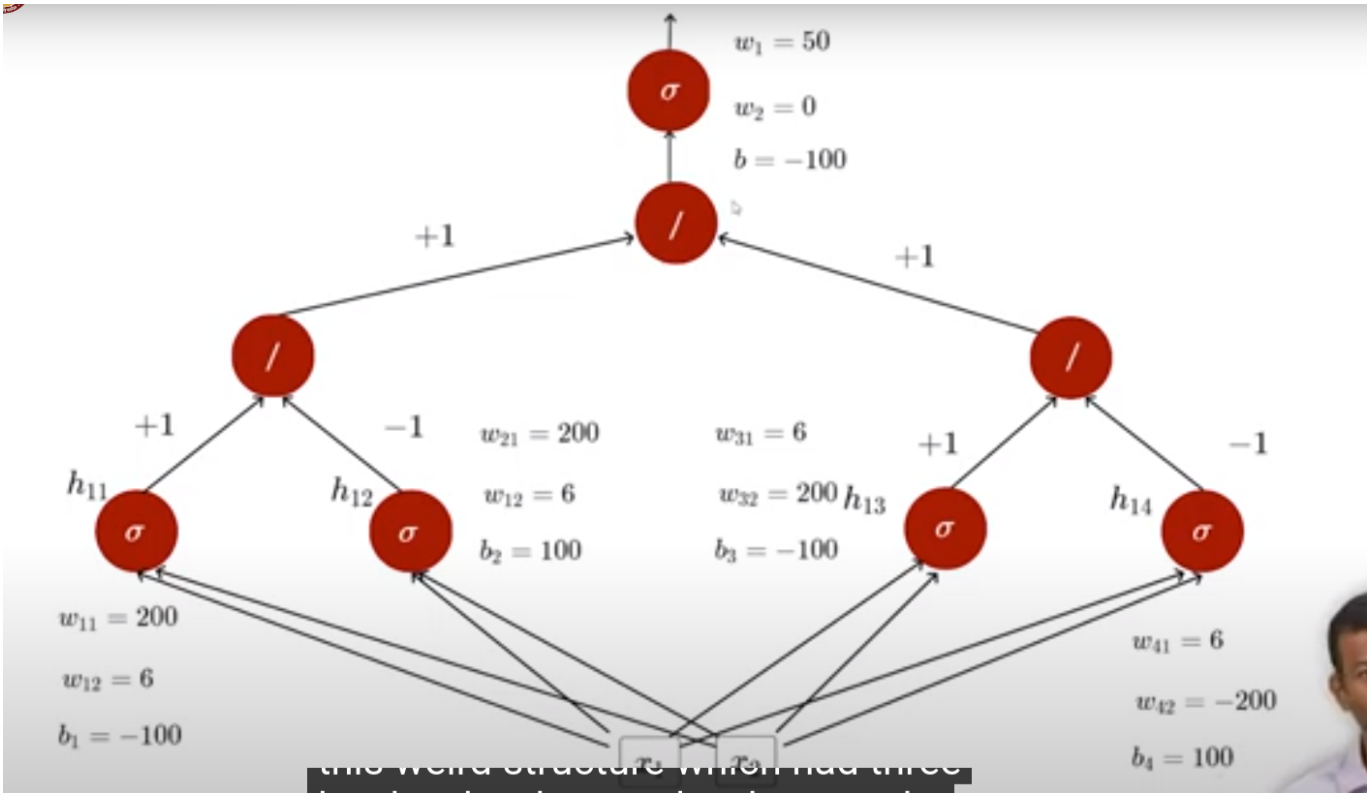


We only need 2 neurons to make a tower, in 2d. and only 1 x. To get a "closed tower in 3d we use 4 neurons.



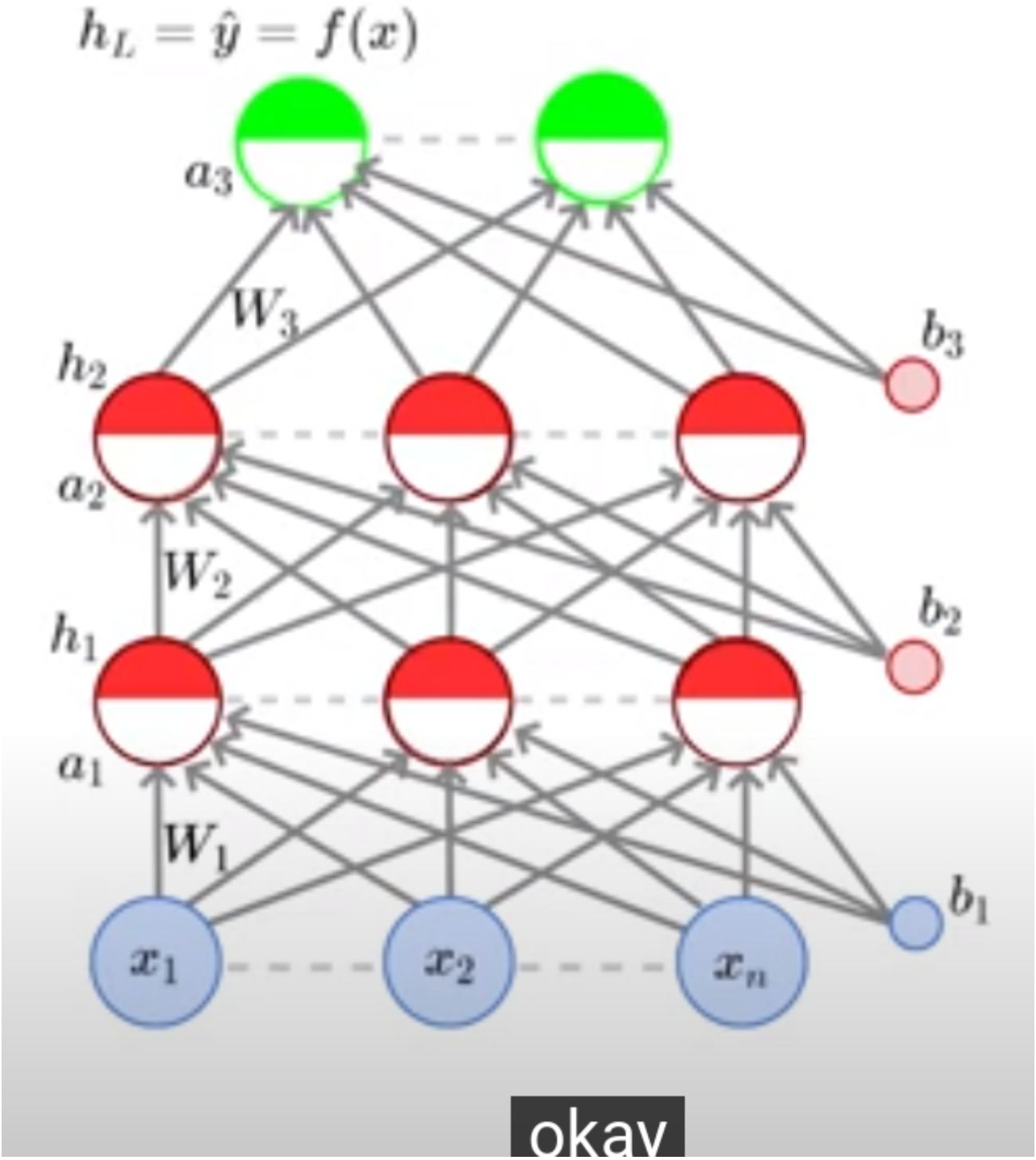
Now if we add this to another sigmoid and set it so the all output above one stays the we get,





Week 3

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



Softmax is

$$\frac{e^{aL_j}}{\sum_{i=1}^{k} e^{aL_j}}$$

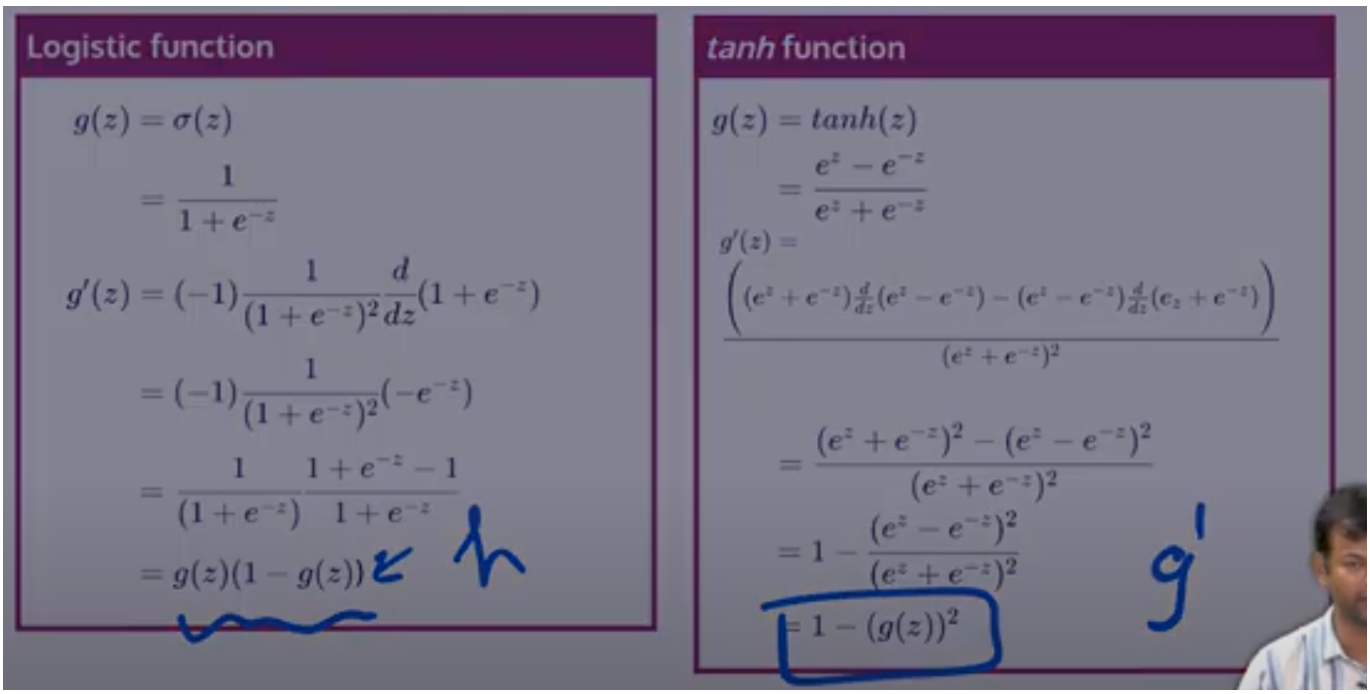
>:(Softmax bounds to 0-1.

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

Now since we use this for probability, the formula simplifies into $l(\theta) = -log\ \hat{y}_l$ where l is the True class.

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

a is value pre-activation function and h is value post-activation function.



Week 4

Momentum based Gradient Descent:

This is mainly made to make GD faster.

formula is

$$u_t = \beta u_{t-1} + \nabla w_t$$
$$u_0 = \nabla w_0$$
$$u_1 = \beta u_o + \nabla w_1$$
$$\dots$$

β is usually less than 1,, so the weightage for thee older changes decreases.

Generally M-GB will overshoot and oscillate back into the solution, but even with this it is still faster than regular GB.

Nesterov Accelerated Gradient Descent

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

Essentially we move by βu_{t-1} and then we calculate w based on the new position, this helps in reducing momentum in the incorrect direction.

Stochastic GD:

Stochastic uses only one point. This is stupid so we used batches instead. Instead of taking all points to make GD we use as few points as we can. This makes it **much cheaper than** GD???. Higher k is more accurate it will be. k in programs are usually called `mini_batch_size`. This can be added onto NAG and MGD.

Now rather than these if we change learning rate, we run into the same oscillation problem, this can be solved by annealing the learning rate. One way s step delay, which halves after every 5 or so epochs. We can also cap the momentum.

Line search evaluates a bunch of learning rates.

For stochastic number of epochs x number of points

For line search just number of epochs

Week 5

A sparse input does not imply that the input is unimportant. So we should change learning rate for these as well. All of the above are not very good when it comes to sparse inputs. They will eventually reach the correct conclusion but it will take more epochs/effort than desired.

Consider weights and inputs as vectors/matrix from no on.

AdaGrad

$$v_t = v_{t-1} + \nabla w_t^2$$
$$w_{t+1} = w_t - \frac{\eta}{\sqrt{v_t + \epsilon}} * \nabla w_t$$

While adagrad is more "optimized" for sparse, MGD and NAG are still faster.

We want $\frac{\eta}{\sqrt{v_t+\epsilon}}$ to "decay" so that the sparse values can more effectively add to the history.

AdaGrad does this effectively and sometimes too effectively. When this goes too far, non-sparse factors start to slow down in unintentional ways. DOES NOT OSCILLATE AROUND MINIMA.

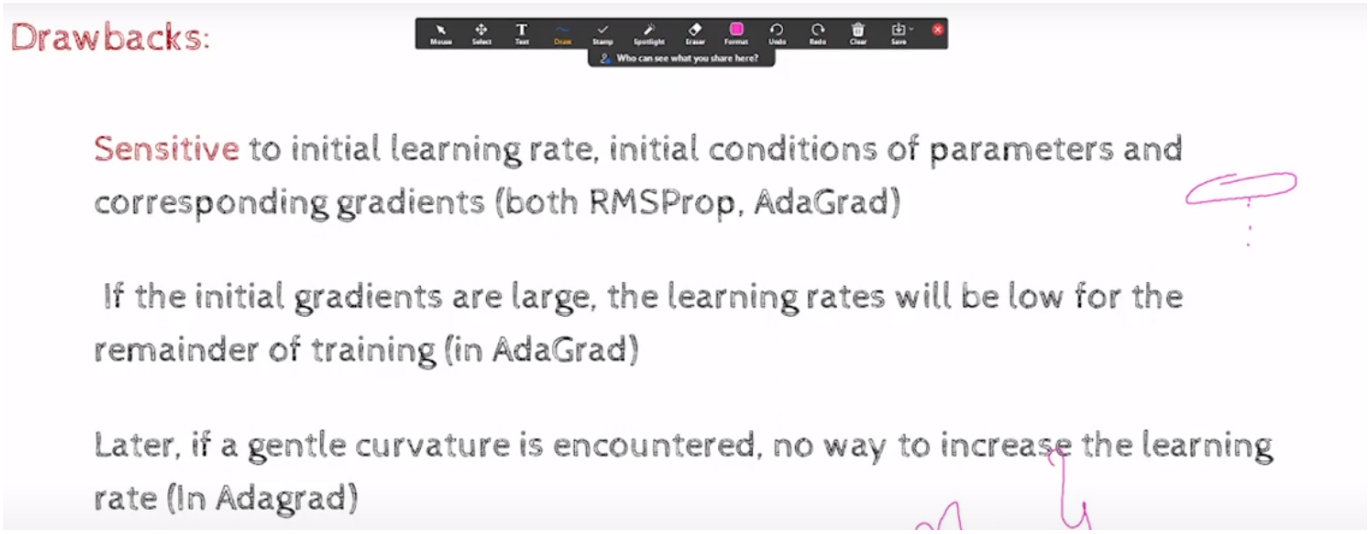
This is solved with RMSProp

RMSProp

The change is we reduce the ability for the history to grow. This is done by a small change in the formula

$$v_t = \beta v_{t-1} + (1-\beta)\nabla w_t^2$$
$$i.e.$$
$$v_t = (1-\beta)\sum_{T=0}^{t} \beta^{t-T}\nabla b_t^2$$

β is between 0 and 1. So this exponentially decays the history. RMSProp is *much* quicker, however it can oscillate around the minima and can potentially never reach the optimal position. It oscillates because of the exponential decay of the history so the learning rate can become constant. AdaGrad does not have this issue, RMSProp can avoid this b varying the initial learning rate.



AdaDelta

This avoids setting an initial η_0. This uses the ratio of 2 histories as the learning rate.

$$for \quad t \quad in \quad range(1, N):$$

1. $\rightarrow \nabla w_t$
2. $\rightarrow v_t = \beta v_{t-1} + (1-\beta)(\nabla w_t)^2$
3. $\rightarrow \Delta w_t = -\frac{\sqrt{u_{t-1}+\epsilon}}{\sqrt{v_t+\epsilon}} \nabla w_t$
4. $\rightarrow w_{t+1} = w_t + \Delta w_t$
5. $\rightarrow u_t = \beta u_{t-1} + (1-\beta)(\Delta w_t)^2$

WARNING THERE IS DELTA Δ AND NABLA ∇ HERE

u_t effectively has older history.

This video has all the above algorithms going with the same parameters.

AdaDelta is the fastest by far.

Adam

This is adaptive movements.

$$m_t = \beta_1 m_{t-1} + (1-\beta_1)\nabla w_t$$
$$\hat{m}_t = \frac{m_t}{1-\beta_1^t}$$
$$v_t = \beta_2 v_{t-1} + (1-\beta_2)(\nabla w_t)^2$$
$$\hat{v}_t = \frac{v_t}{1-\beta_2^t}$$
$$w_{t+1} = w_t - \frac{\eta}{\sqrt{\hat{v}_t}+\epsilon} m_t$$

The hat thing is called bias correction. This is done to make sure that the initial learning rates are not ridiculous.

L normalization

$$L^p = (|x_1|^p + |x_2|^p + ... + |x_n|^p)^{\frac{1}{p}}$$

Adamax

This uses the max of the recent few of the history.

NAdam

Nesterov Adam. Same concept as the previous Nesterov.

Update Rule for NAdam

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

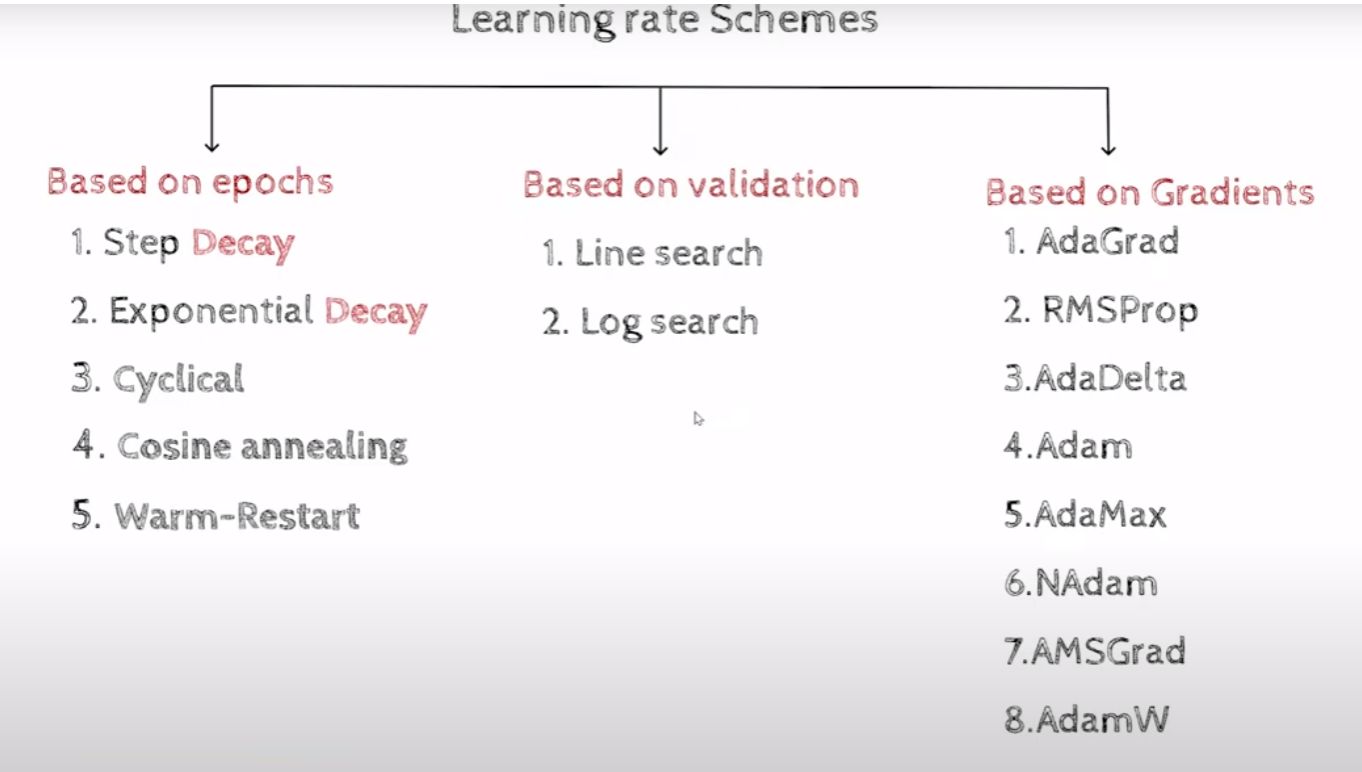
$$\hat{m}_{t+1} = \frac{m_{t+1}}{1-\beta_1^{t+1}}$$

$$v_{t+1} = \beta_2 v_t + (1 - \beta_2)(\nabla w_t)^2$$

$$\hat{v}_{t+1} = \frac{v_{t+1}}{1-\beta_2^{t+1}}$$

$$w_{t+1} = w_t - \frac{\eta}{\sqrt{\hat{v}_{t+1}} + \epsilon}(\beta_1 \hat{m}_{t+1} + \frac{(1 - \beta_1)\nabla w_t}{1 - \beta_1^{t+1}})$$

Learning rate schemes

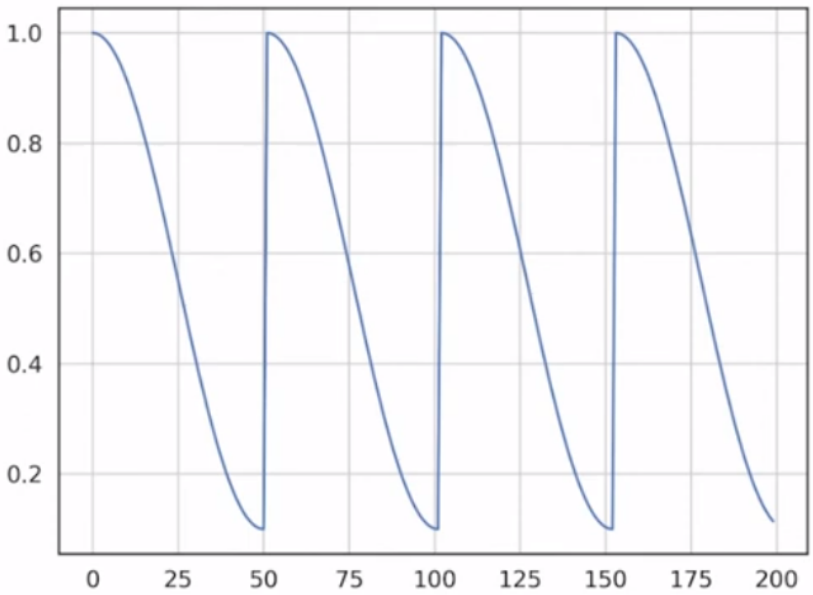


Cyclic

There are chances that the model settles in a saddle point..In this case,, often that minima is not escapeble because of a low learning rate, so what cyclic does is it sets a min η and max η , and it increases in a cyclic motion. Usually a triangle signal.

Cosine aneeling

similar to above but shape is like below. This is called warm restart because of the jump.



Week 6

Bias and variance

Over parameterized models tend to overfit and are sensitive to training data. (Some even have more parameters than data points)

Bias here is the difference between the average produced value and average expected value.

Variance is the difference between different runs of the same model. $\hat{f}(x) = E[(\hat{f}(x) - E[\hat{f}(x))^2]$ this is standard deviation formula it seems.

Model complexity tends to always reduce training error. On test data there is a sweet spot and beyond that the test error start increasing again (Overfit)

True error and model complexity:

$\Omega(\theta)$ is empirical training error, it increases as the training error decreases. We use normalization/ regularization to solve this issue.

Regularization

L2 Regularization

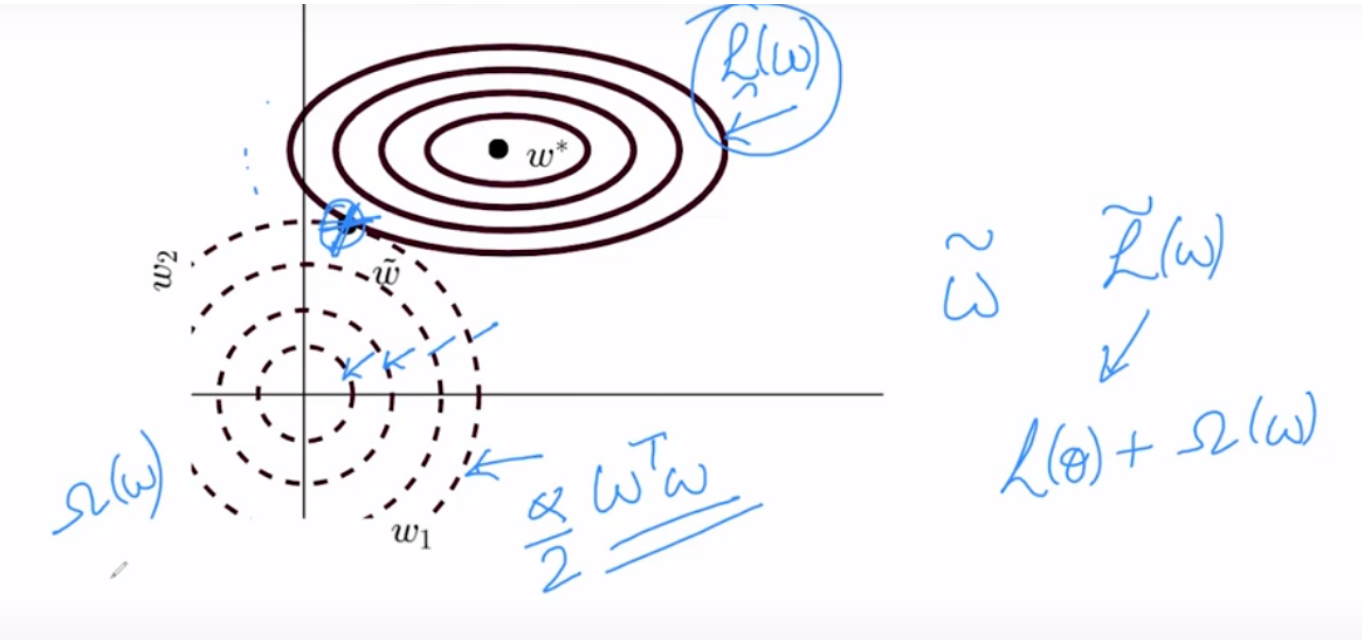
$\tilde{L}(w) = L(w) + \frac{\alpha}{2}||w||^2$

This kinda sets a boundary for potential values for w.

For SGD:

$w_{t+1} = w_t - \eta\nabla L(w_t) - \eta\alpha w_t$

update rule is above.



The brow line are increasing loss values, we can see w_1 decreases a lot more than w_2. This is because a change in w_1 has a much smaller impact on loss. Look at it like a contour map.

L2 tends to increase training error and decrease testing error. it also reduces complexity by making unimportant parameters tend towards 0.

Dataset Augmentation

This is another regularization method. This is something like rotating an image a bit or shifting etc. This is also to prevent overfitting. Also works for speech.

Noise addition to inputs

This helps the network be more flexible.

Noise addition to outputs

We replace hard targets with soft targets. So if we have [0,0,1,0] we replace with $\left[\frac{\epsilon}{4}, \frac{\epsilon}{4}, 1-\epsilon, \frac{\epsilon}{4}\right]$ where ϵ is a small positive constant.

Early stopping

We check if validation error is reducing. Once it stagnates, we go back to the lowest validation error model we had. I allows "t" changes to parameters.

have heart attack :) 🩶

$$w_t = (I - \eta H)w_{t-1} + \eta H w^*$$

Using EVD of H as $H = Q\Lambda Q^T$, we get:

$$w_t = (I - \eta Q\Lambda Q^T)w_{t-1} + \eta Q\Lambda Q^T w^*$$

If we start with $w_0 = 0$ then we can show that (See Appendix)

$$w_t = Q[I - (I - \varepsilon\Lambda)^t]Q^T w^*$$

Compare this with the expression we had for optimum $\tilde{W}$ with L_2 regularization

$$\tilde{w} = Q[I - (\Lambda + \alpha I)^{-1}\alpha]Q^T w^*$$

Ignore this lol.

Ensemble

Ensemble is taking votes from different models.

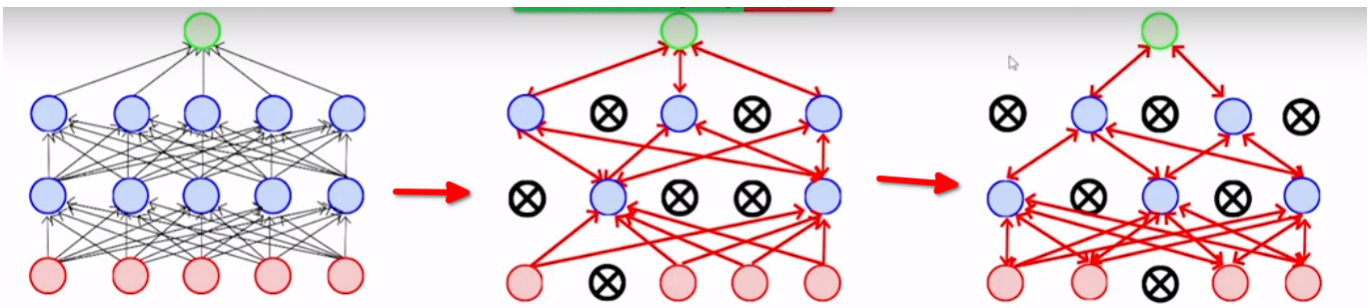
Bagging

Same model, different dataset.

Dropout

Let's say a neural network has n nodes, we drop m nodes from the network and train tat network. Each node has a probability of being dropped out.

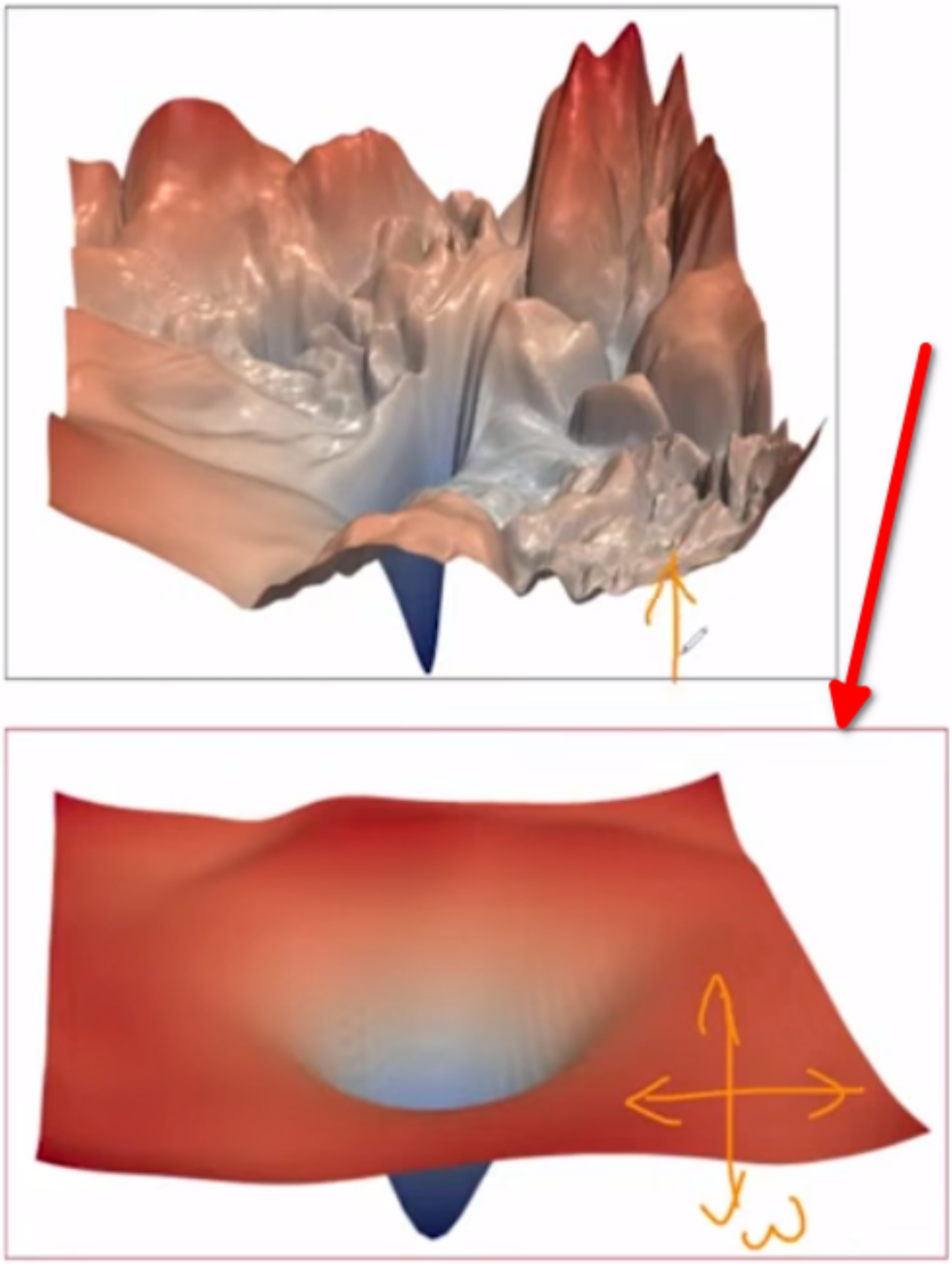
This is done for each epoch, m nodes participate in forward and backward propagation.



We can optionally weight the weights with the probability.

Dropout prevents co-adopting.

Below is what regularization does



$$\mathscr{L}(\theta) = \frac{1}{N}\sum_{i=1}^{N} l\big(f(x_i,\theta), y\big)$$

Data x_i

1. Data Augmentation
2. Noise Injection

Architecture choice $f(\cdot)$

1. Dropout
2. Skip connections (CNN)
3. Weight sharing
4. Pooling

Optimizer ∇

1. SGD
2. Early stopping

Penalize cost $\mathscr{L}(\cdot)$

1. L_1
2. L_2

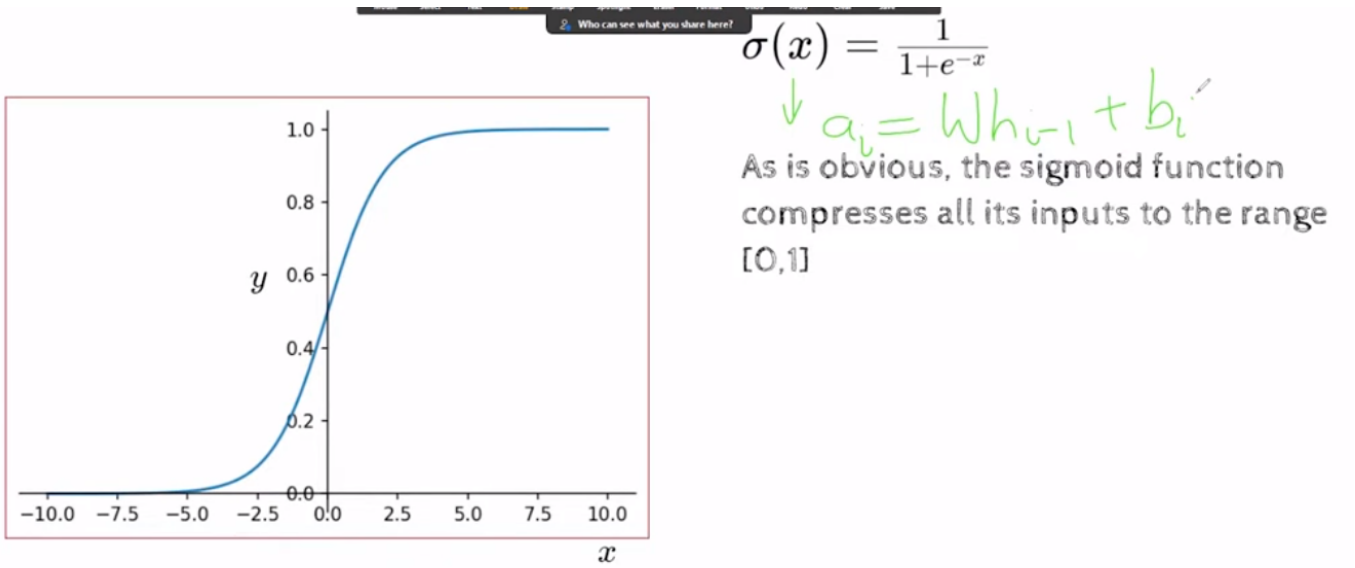
Week 7

bro pre-training helps no matter the size of the network.

If we can capture the logic of n neurons in just m neurons then bueno :)

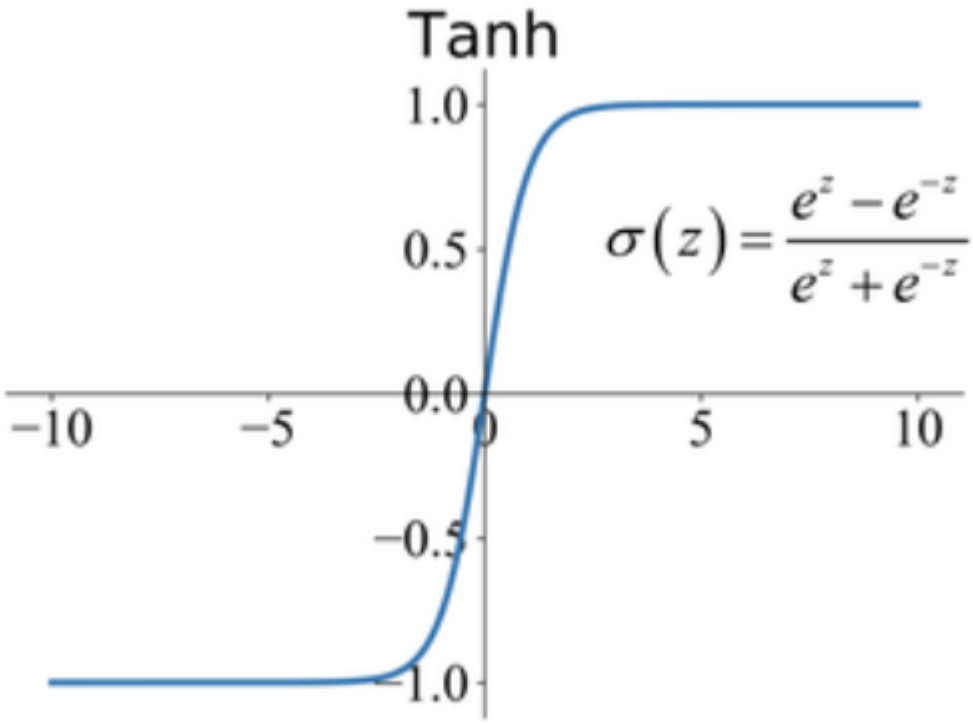
Activation functions

Sigmoid



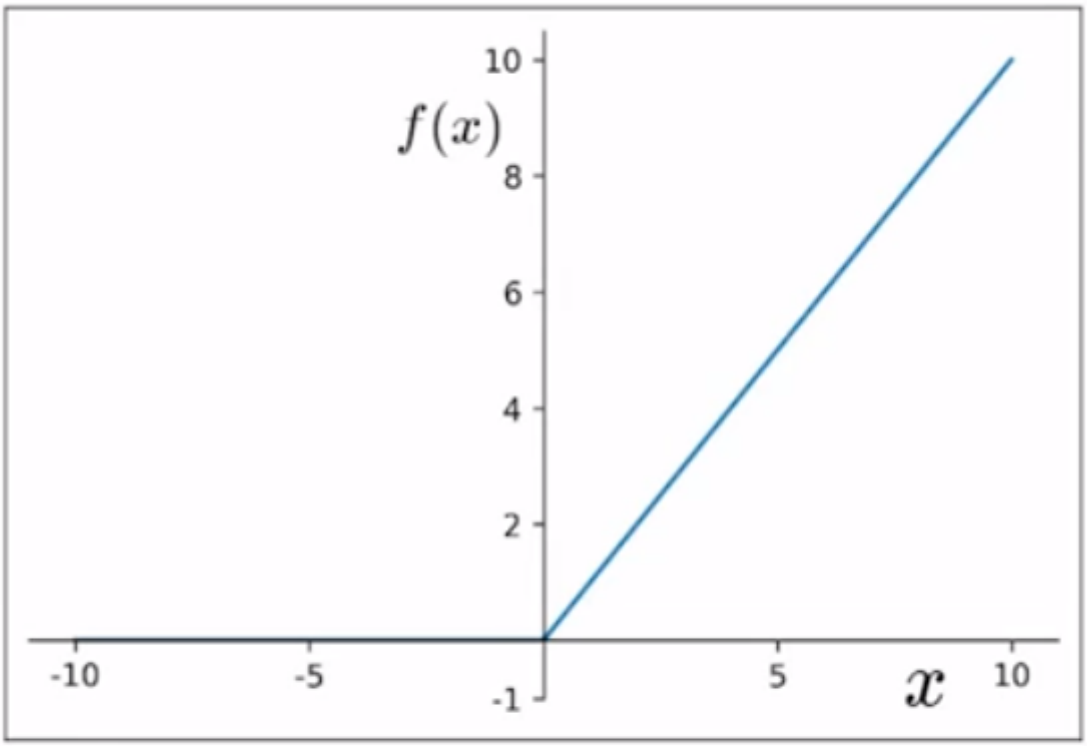
At large x values, the gradient vanishes. It is also not 0 centered,it is 0.5 centered. So it does not allow negitive values which reduces the directions.

tanh



same but 0 centered.

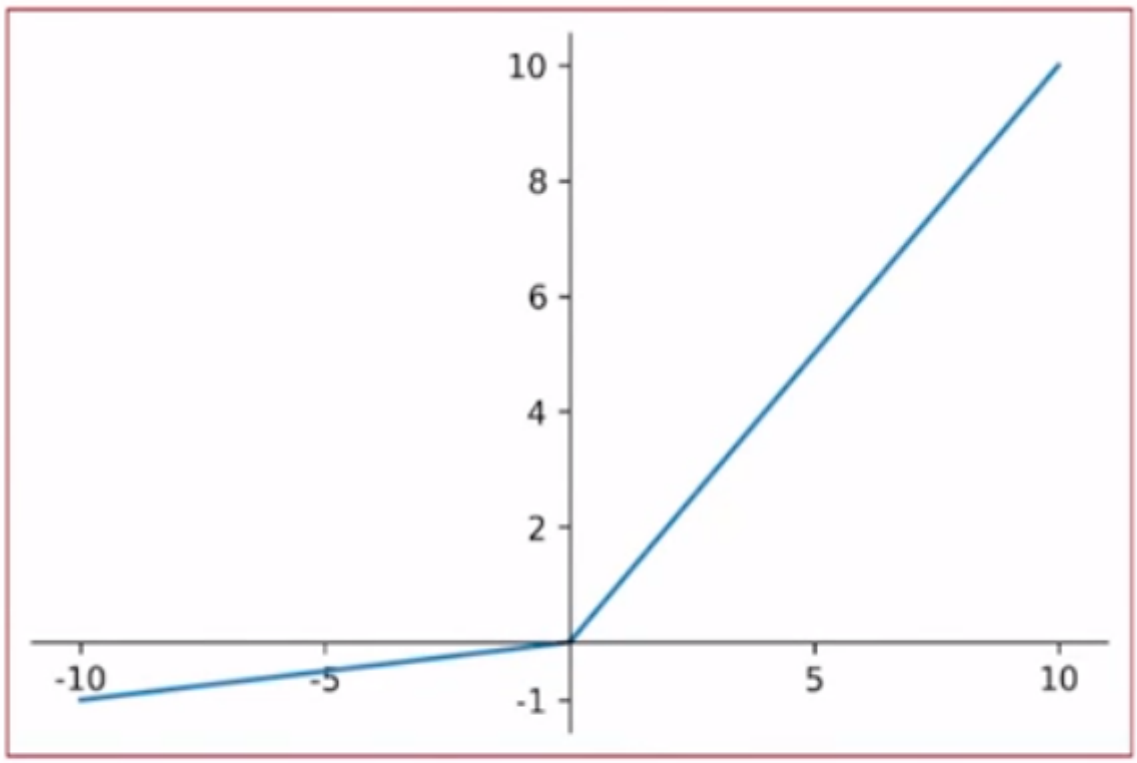
ReLU



$$f(x) = max(0, x)$$

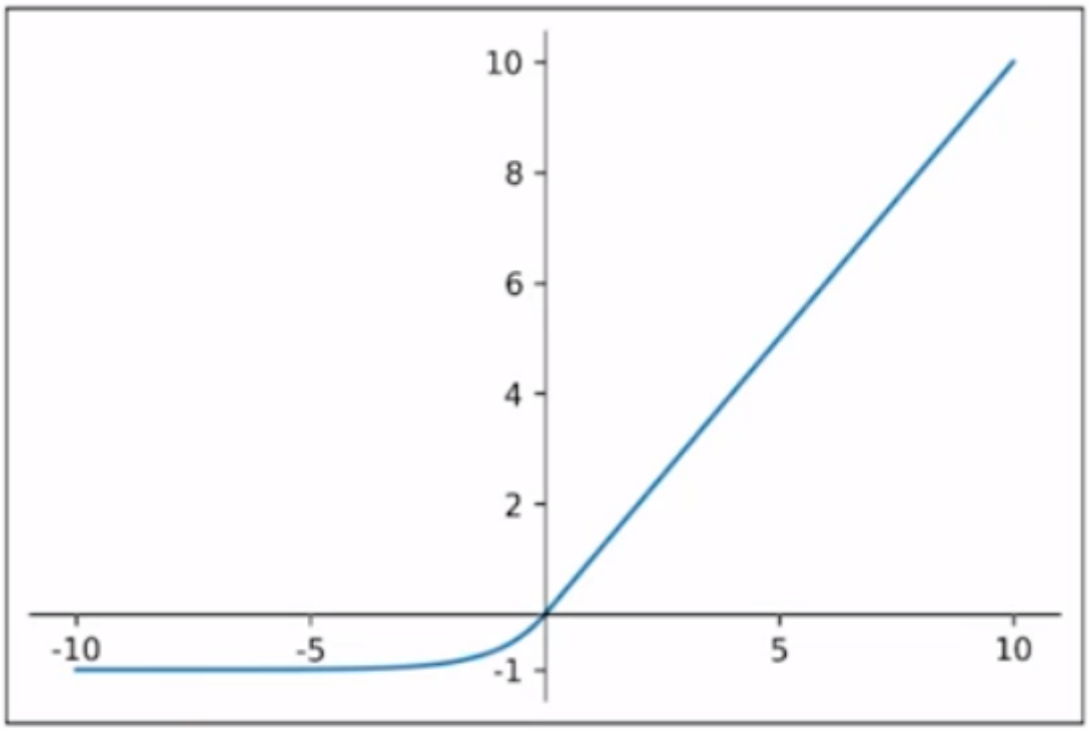
We can combine 2 relus o emulate sigmoid. This does not saturate in positive, very efficient but same direction issue. Also negative ignored is not the best idea. Can cause dead neurons.

Leaky ReLU



$$f(x) = max(0.1x, x)$$

Exponential linear Unit (ReLU)



$$f(x) = x \quad if \quad x > 0$$

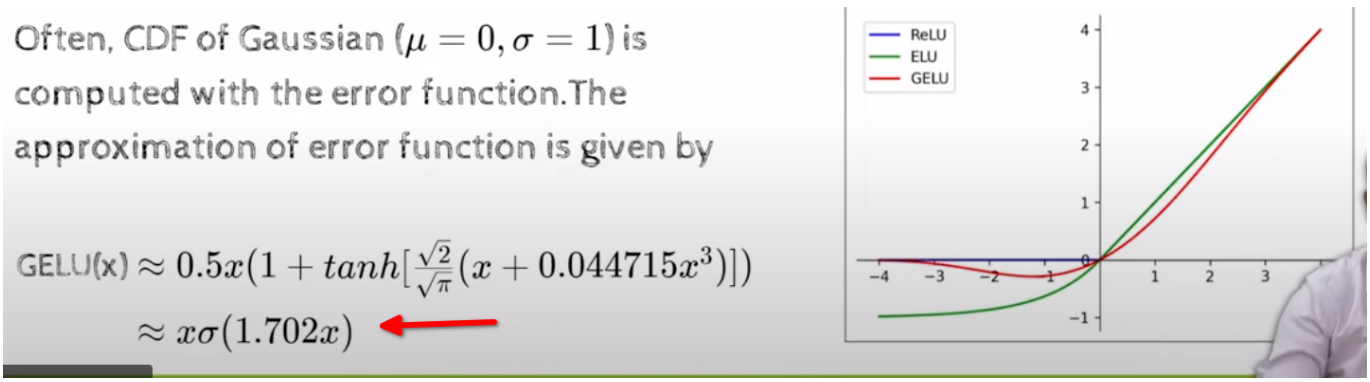
$$= ae^{x} - 1 \quad if \quad x \leq 0$$

MaxOut

We take max of m neurons and keep only the max.

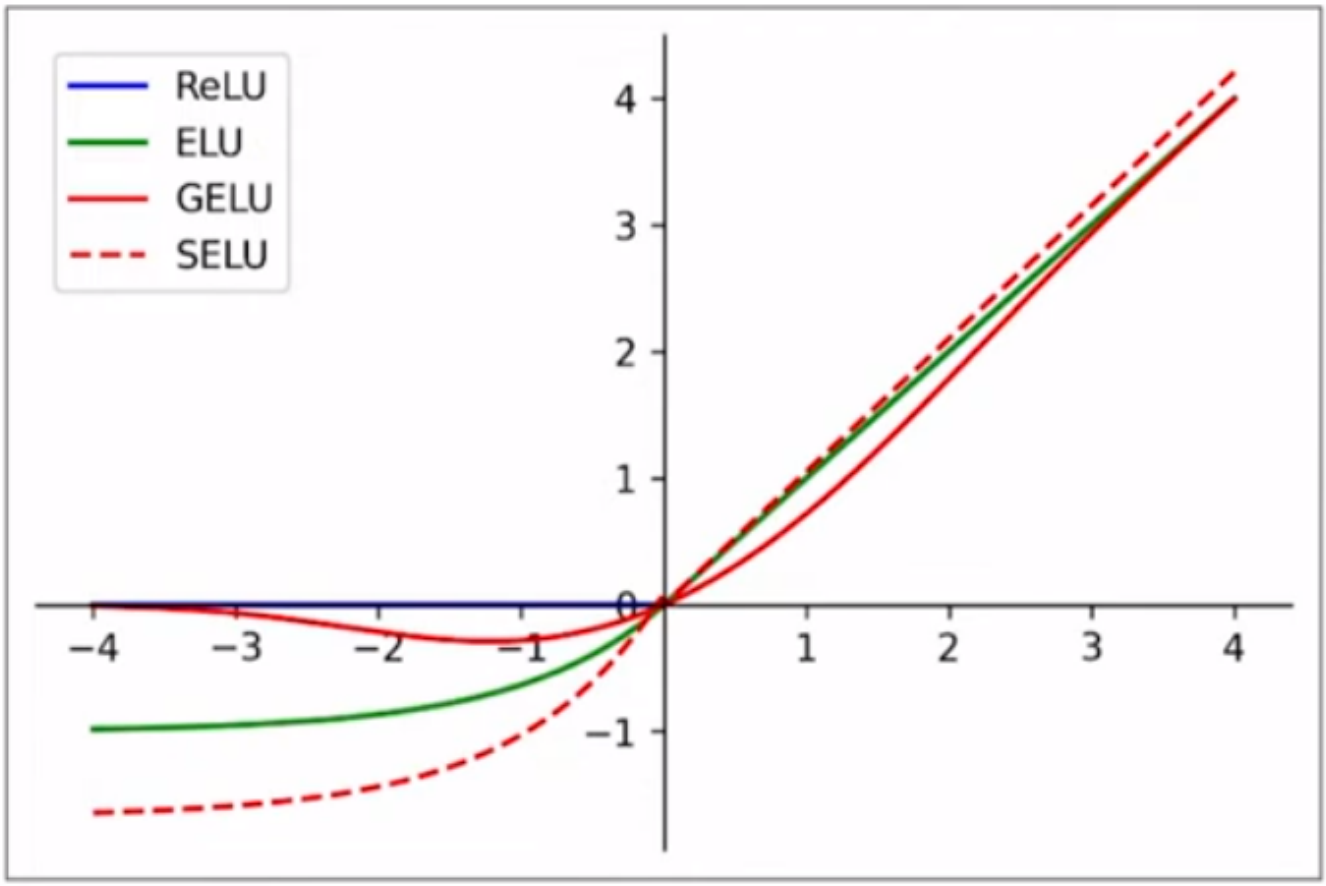
GELU

Gaussian Error Linear Unit retarded formula.



SeLU

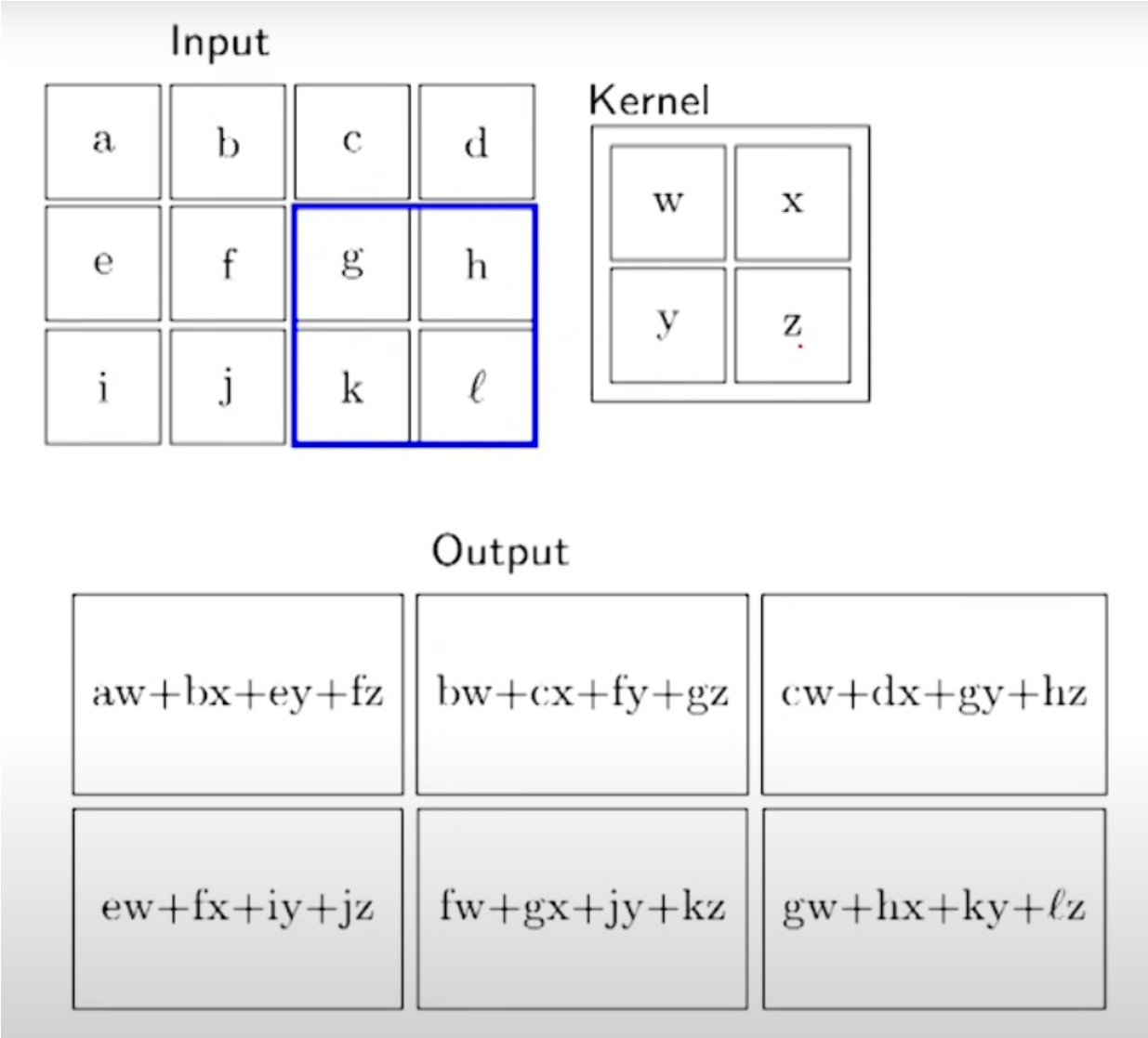
Scaled Exponential Linear Unit. This tries to be 0 centered.



Eventually we settled on getting a chain of functions. SWISH is a type of GeLU.

Week 8

Traditional convolution take all previous readings, in practice we keep a window. Weighted average of the neighbors for image.



Relation between input output and filter size

kernel = filter

Width and height reduce by F - 1.

$W_2 = \frac{W_1 - F + 2P}{S} + 1$

"Depth" = Number of filters

The advantage of CNN is the filter, a normal NN will take inputs from all the pixels, in CNN we can make it so that one neuron only takes that filters input. CNN has weight sharing cuz kernel is same. This causes very few weights so underfitting so we use multiple kernels.

We do input -> filter -> pooling -> filter ... -> Fully Connected Layer

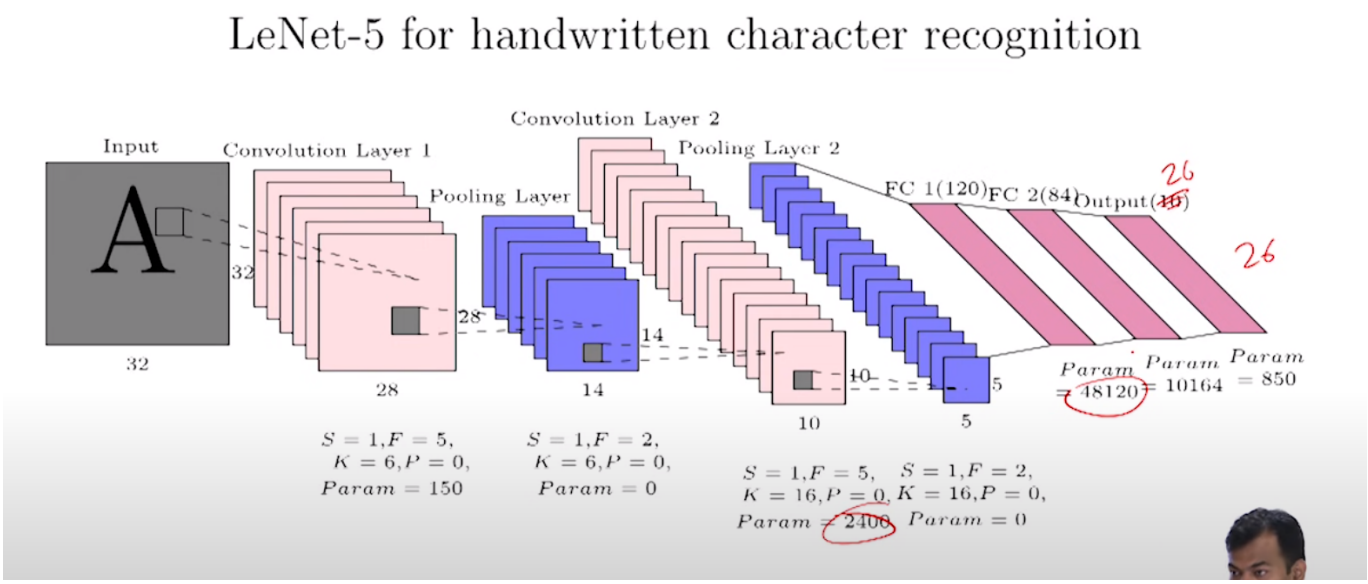
Number of params = FxFxKxM (K is num of filters, M is depth of input)

Number of params in pooling is 0

Fully Connected layer:

HxWxD

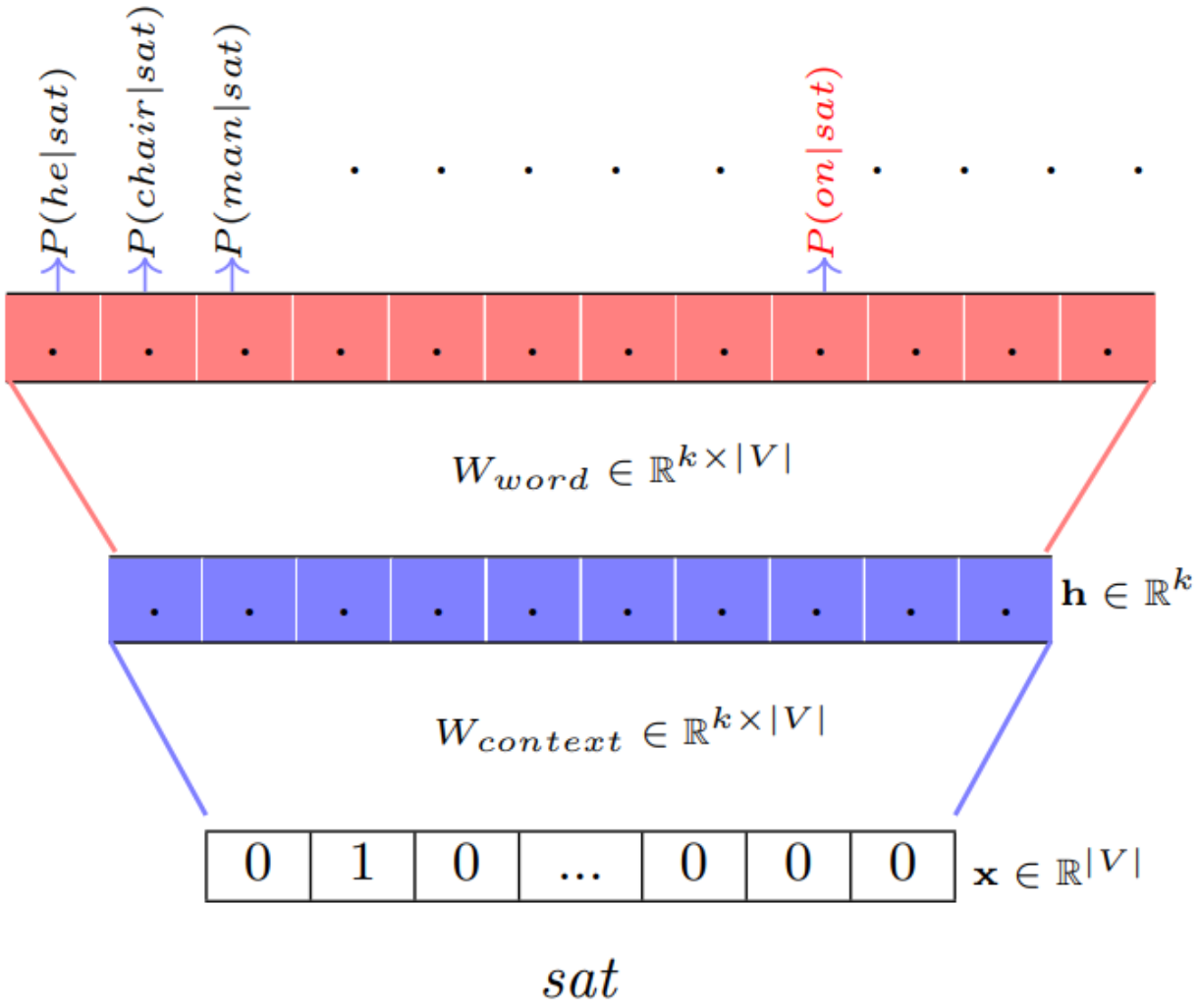
from here on it is a regular network.



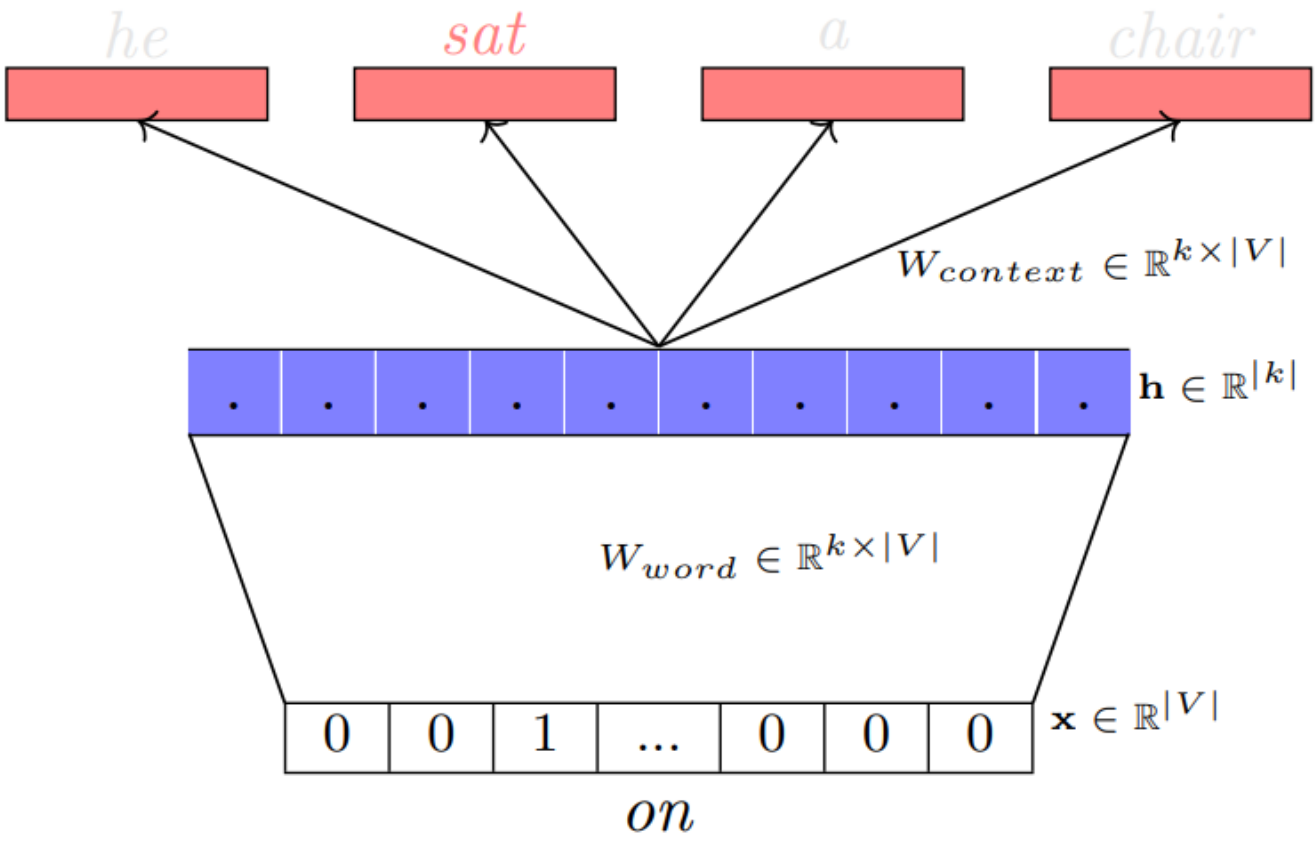
Week 9

Some SVD stuff ignored.

continuous bag of words model



skip gram model



$$PMI(w,c) = \log \frac{p(c|w)}{p(c)}$$

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

How many weight matrices does a LSTM unit and GRU unit learns respectively during backpropagation?

LSTM 8 & GRU 6

LSTM and GRU handle exploding and vanishing gradient using gating

To find the "link" between words make a co-occurence matrix.

$$PMI(Pointwise\ Mutual\ Information) = log \frac{count(w,c) * N}{count(w) * count(c)}$$

$$Where:$$
$$w\ is\ word\ 1$$
$$c\ is\ word\ 2$$

N *is sum of all words.*