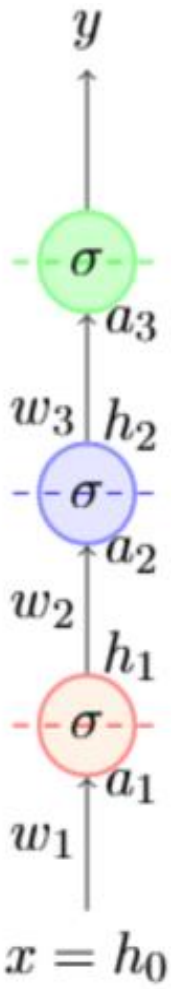
Topics

- Improved Gradient Descent Techniques for Back propagation
- Activation Functions
- Regularization
- Weight Initialisation



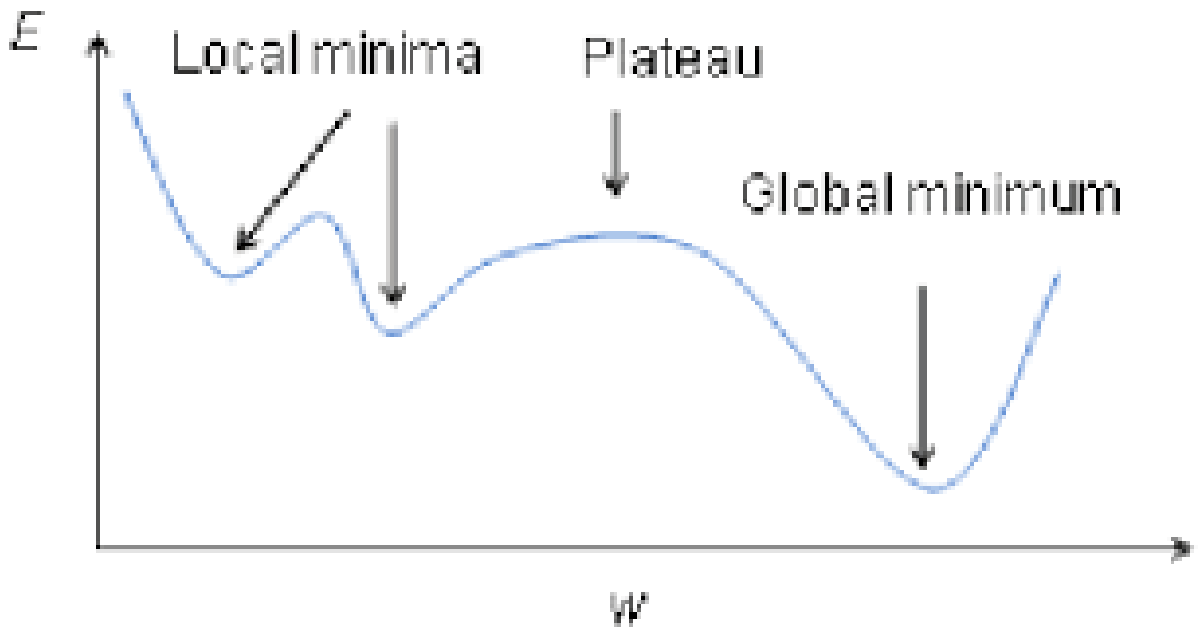
$$\frac{\partial \mathscr{L}(\mathbf{w})}{\partial w_1} = \frac{\partial \mathscr{L}(\mathbf{w})}{\partial y} \cdot \frac{\partial y}{\partial a_3} \cdot \frac{\partial a_3}{\partial h_2} \cdot \frac{\partial h_2}{\partial a_2} \cdot \frac{\partial a_2}{\partial h_1} \cdot \frac{\partial h_1}{\partial a_1} \cdot \frac{\partial a_1}{\partial w_1}$$

$$= \frac{\partial \mathscr{L}(\mathbf{w})}{\partial y} * \ldots\ldots\ldots\ldots * h_0$$

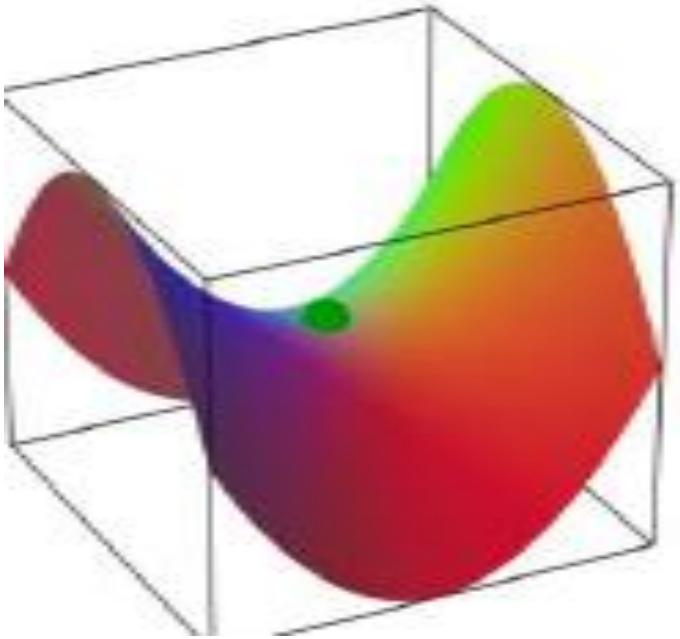
In general,

$$\nabla w_i = \frac{\partial \mathscr{L}(\mathbf{w})}{\partial y} * \ldots\ldots\ldots\ldots * h_{i-1}$$

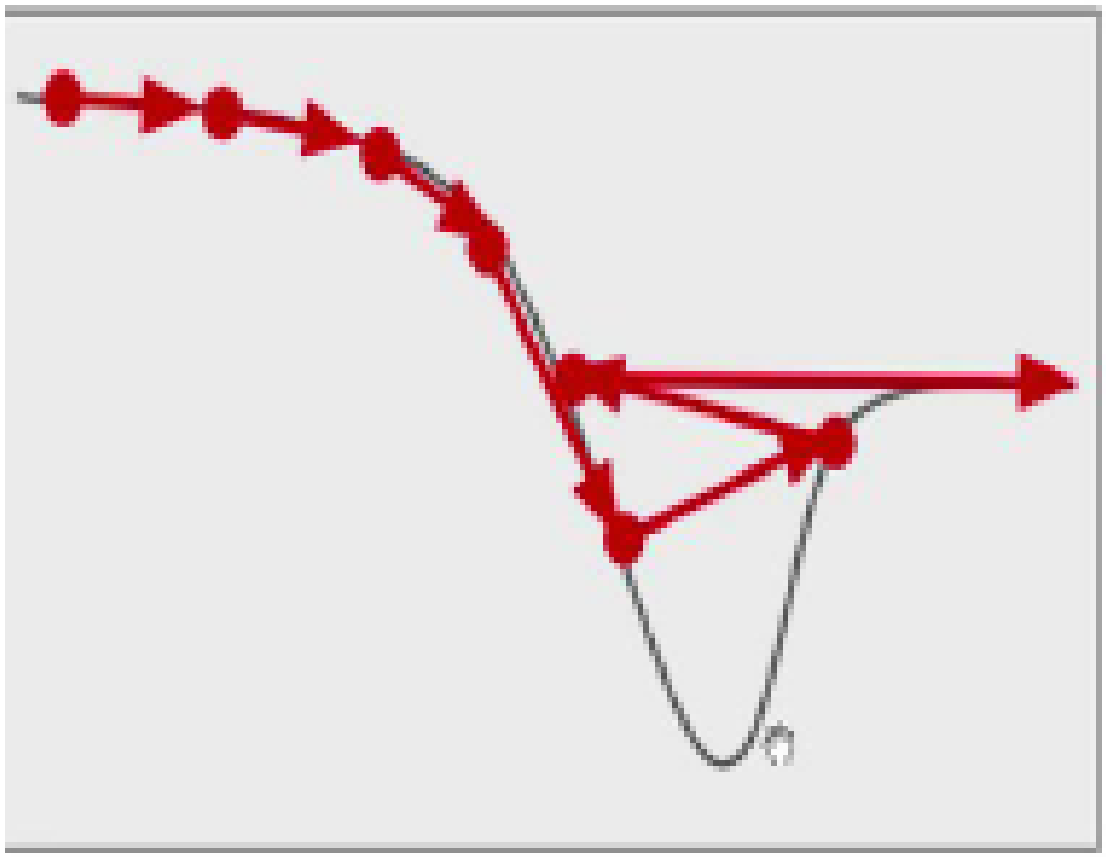
Typical error plot corresponding to one parameter

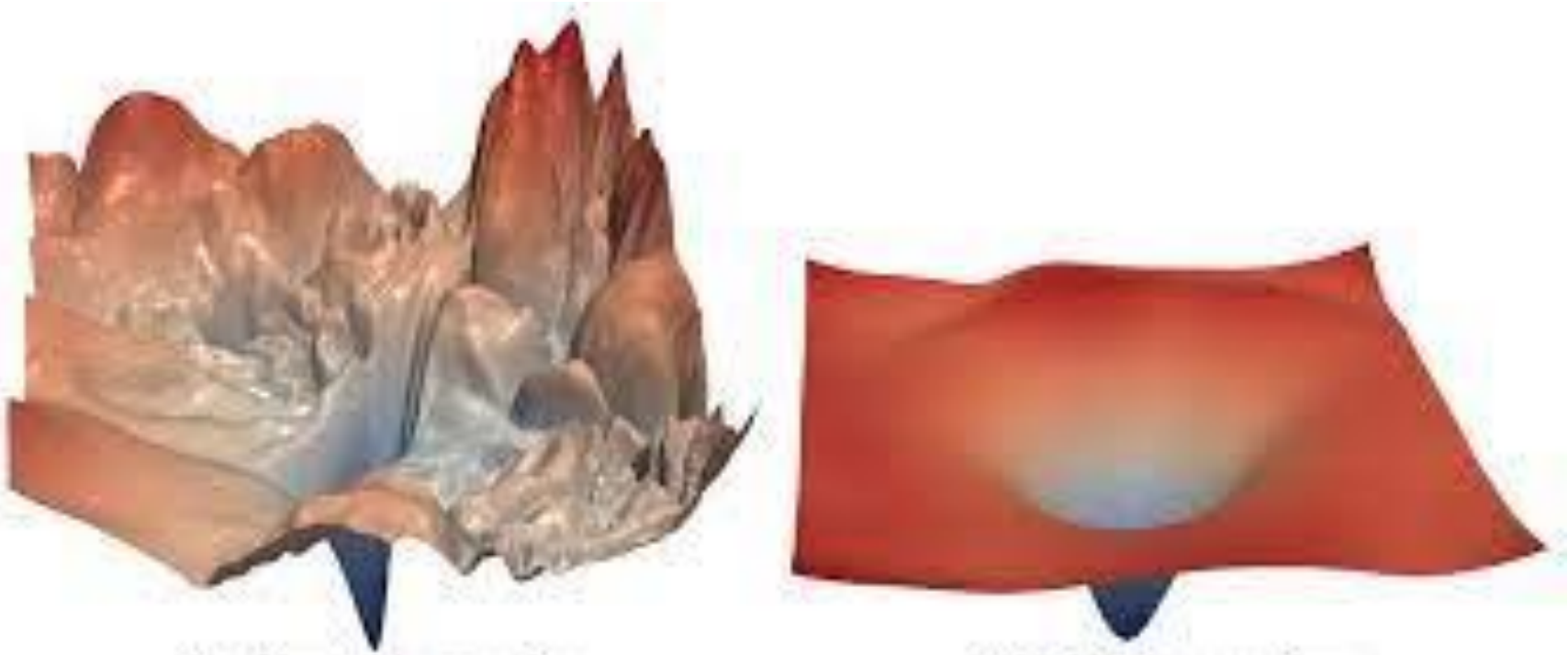


Issues in saddle point



Issues in plateaus





(a) without skip connections

(b) with skip connections

Some observations about gradient descent

- It takes a lot of time to navigate regions having a gentle slope
- This is because the gradient in these regions is very small
- Can we do something better ?
- Yes, let's take a look at 'Momentum based gradient descent'

- Even in the regions having gentle slopes, momentum based gradient descent is able to take large steps because the momentum carries it along

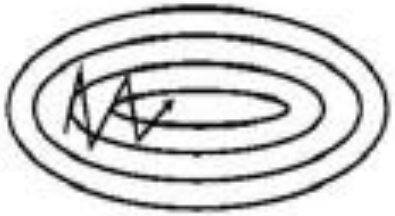
$$update_t = \gamma \cdot update_{t-1} + \eta \nabla w_t$$
$$w_{t+1} = w_t - update_t$$

In addition to the current update, also look at the history of updates.

$$\begin{aligned}
update_0 &= 0 \\
update_1 &= \gamma \cdot update_0 + \eta \nabla w_1 = \eta \nabla w_1 \\
update_2 &= \gamma \cdot update_1 + \eta \nabla w_2 = \gamma \cdot \eta \nabla w_1 + \eta \nabla w_2 \\
update_3 &= \gamma \cdot update_2 + \eta \nabla w_3 = \gamma(\gamma \cdot \eta \nabla w_1 + \eta \nabla w_2) + \eta \nabla w_3 \\
&= \gamma \cdot update_2 + \eta \nabla w_3 = \gamma^2 \cdot \eta \nabla w_1 + \gamma \cdot \eta \nabla w_2 + \eta \nabla w_3 \\
update_4 &= \gamma \cdot update_3 + \eta \nabla w_4 = \gamma^3 \cdot \eta \nabla w_1 + \gamma^2 \cdot \eta \nabla w_2 + \gamma \cdot \eta \nabla w_3 + \eta \nabla w_4 \\
&\vdots \\
update_t &= \gamma \cdot update_{t-1} + \eta \nabla w_t = \gamma^{t-1} \cdot \eta \nabla w_1 + \gamma^{t-2} \cdot \eta \nabla w_1 + \ldots + \eta \nabla w_t
\end{aligned}$$

We consider choosing value for γ in (0,1). Typically, it is set to 0.9.

Without momentum



With momentum

- Damps step sizes along directions of high curvature, yielding a larger effective learning rate along the directions of low curvature.
- Larger the γ, more the previous gradients affect the current step.
- Generally, γ is set to 0.5 until initial learning stabilizes and then increased to 0.9 or higher (practitioners often default it to 0.9/0.95 these days)

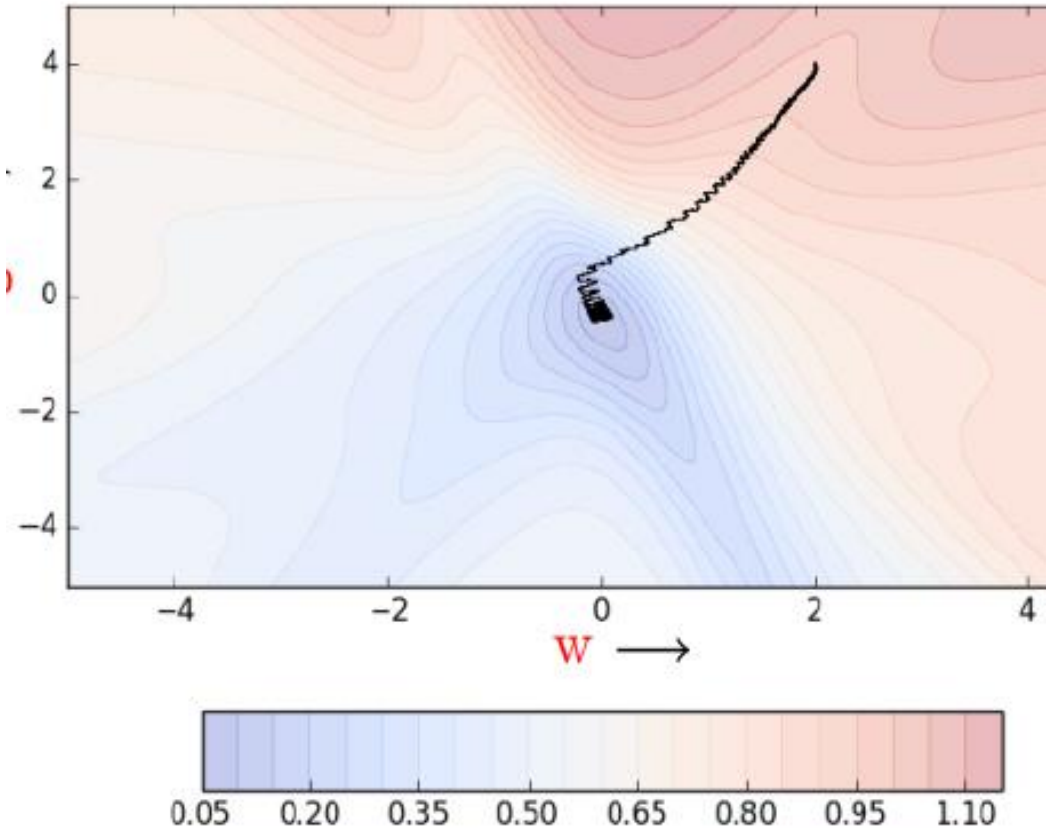
Nesterov Accelerated Gradient Descent

- Look before you leap
- Recall that $update_t = \gamma \cdot update_{t-1} + \eta \nabla w_t$
- So we know that we are going to move by at least by $\gamma \cdot update_{t-1}$ and then a bit more by $\eta \nabla w_t$
- Why not calculate the gradient (∇w_{look_ahead}) at this partially updated value of w $(w_{look\_ahead} = w_t - \gamma \cdot update_{t-1})$ instead of calculating it using the current value w_t

$$
\begin{aligned}
w_{look\_ahead} &= w_t - \gamma \cdot update_{t-1} \\
update_t &= \gamma \cdot update_{t-1} + \eta \nabla w_{look\_ahead} \\
w_{t+1} &= w_t - update_t
\end{aligned}
$$

Stochastic Gradient Descent

- Updation of parameters for every single point.
- No guarantee that loss will reduce at each iteration



Mini Batch stochastic gradient

- Updation of parameters by considering mini batch of data points
- Better estimate compared to conventional stochastic gradient

Tips for initial learning rate ?

- Tune learning rate [Try different values on a log scale: 0.0001, 0.001, 0.01, 0.1. 1.0]
- Run a few epochs with each of these and figure out a learning rate which works best
- Now do a finer search around this value [for example, if the best learning rate was 0.1 then now try some values around it: 0.05, 0.2, 0.3]
- Disclaimer: these are just heuristics ... no clear winner strategy

Tips for annealing learning rate

- **Step Decay:**
  - Halve the learning rate after every 5 epochs or
  - Halve the learning rate after an epoch if the validation error is more than what it was at the end of the previous epoch
- **Exponential Decay:** $\eta = \eta_0^{-kt}$ where η_0 and k are hyperparameters and t is the step number
- **1/t Decay:** $\eta = \frac{\eta_0}{1+kt}$ where η_0 and k are hyperparameters and t is the step number

Line Search

- In practice, often a line search is done to find a relatively better value of learning rate.
- Update **w** using different values of learning rate.
- Now retain that updated value of **w** which gives the lowest loss
- Essentially at each step we are trying to use the best value from the available choices
- We are doing many more computations in each step

Adaptive gradients

Intuition

- Decay the learning rate for parameters in proportion to their update history (more updates means more decay)

Update rule for Adagrad

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

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

... and a similar set of equations for b_t

Intuition

- Adagrad decays the learning rate very aggressively (as the denominator grows)
- As a result after a while the frequent parameters will start receiving very small updates because of the decayed learning rate
- To avoid this why not decay the denominator and prevent its rapid growth

Update rule for RMSProp

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

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

Intuition

- Do everything that RMSProp does to solve the decay problem of Adagrad
- Plus use a cumulative history of the gradients
- In practice, $\beta_1 = 0.9$ and $\beta_2 = 0.999$

Update rule for Adam

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

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

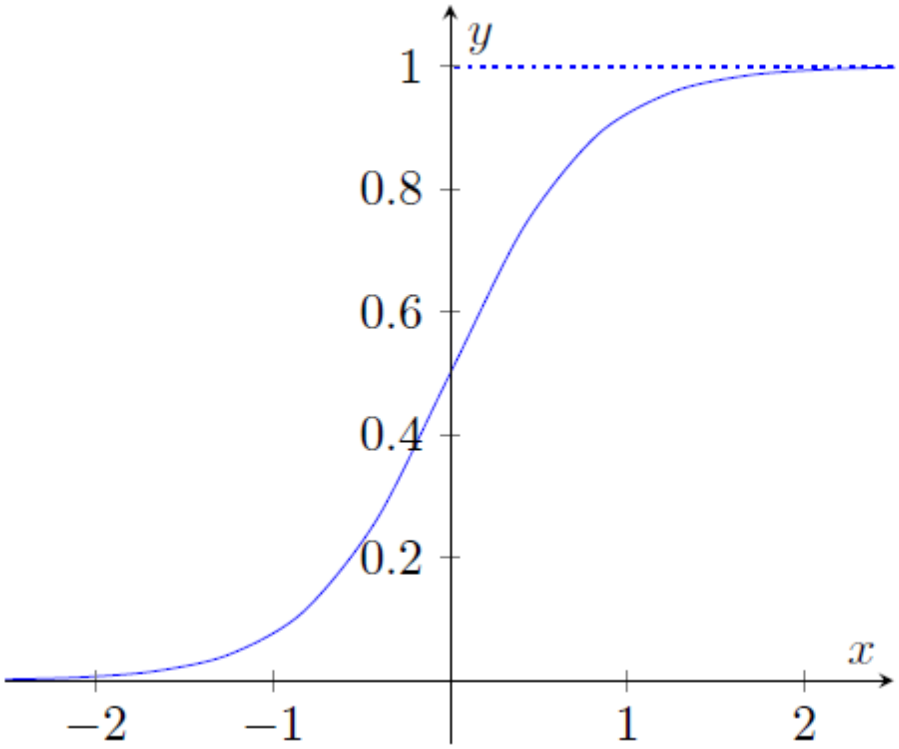
$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t} \qquad \hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

$$w_{t+1} = w_t - \frac{\eta}{\sqrt{\hat{v}_t + \epsilon}} * \hat{m}_t$$

Million dollar question: Which algorithm to use in practice

- Adam seems to be more or less the default choice now ($\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 1e-8$)
- Although it is supposed to be robust to initial learning rates, we have observed that for sequence generation problems $\eta = 0.001, 0.0001$ works best
- Having said that, many papers report that SGD with momentum (Nesterov or classical) with a simple annealing learning rate schedule also works well in practice (typically, starting with $\eta = 0.001, 0.0001$ for sequence generation problems)
- Adam might just be the best choice overall!!

Activation Functions :



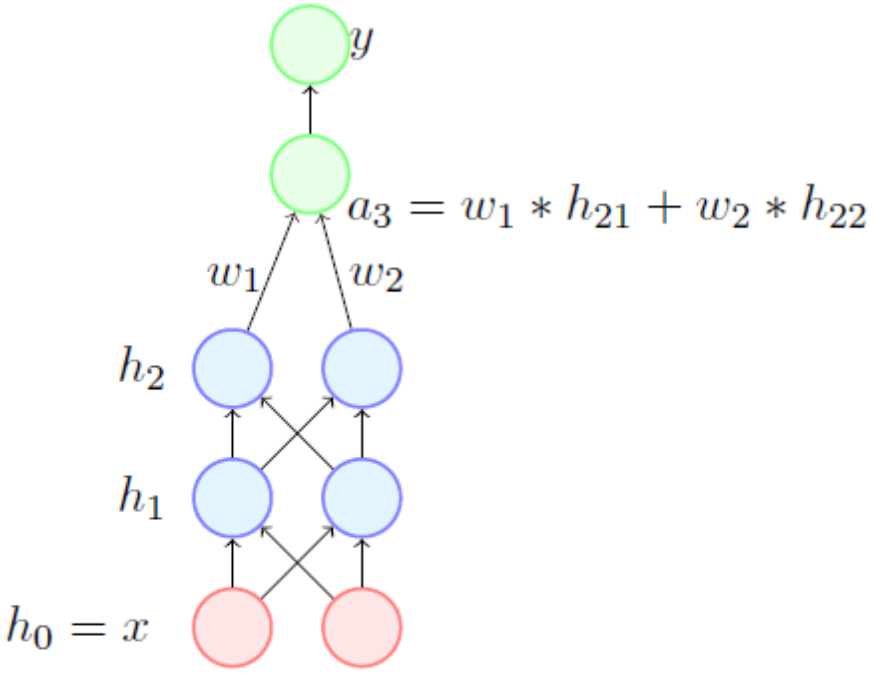
- A sigmoid neuron is said to have saturated when $\sigma(x) = 1$ or $\sigma(x) = 0$

Saturated neurons thus cause the gradient to vanish.

- Saturated neurons cause the gradient to vanish
- Sigmoids are not zero centered
- Consider the gradient w.r.t. w_1 an w_2

$$\nabla w_1 = \frac{\partial \mathscr{L}(\mathbf{w})}{\partial y} \frac{\partial y}{h_3} \frac{\partial h_3}{\partial a_3} \quad h_{21}$$

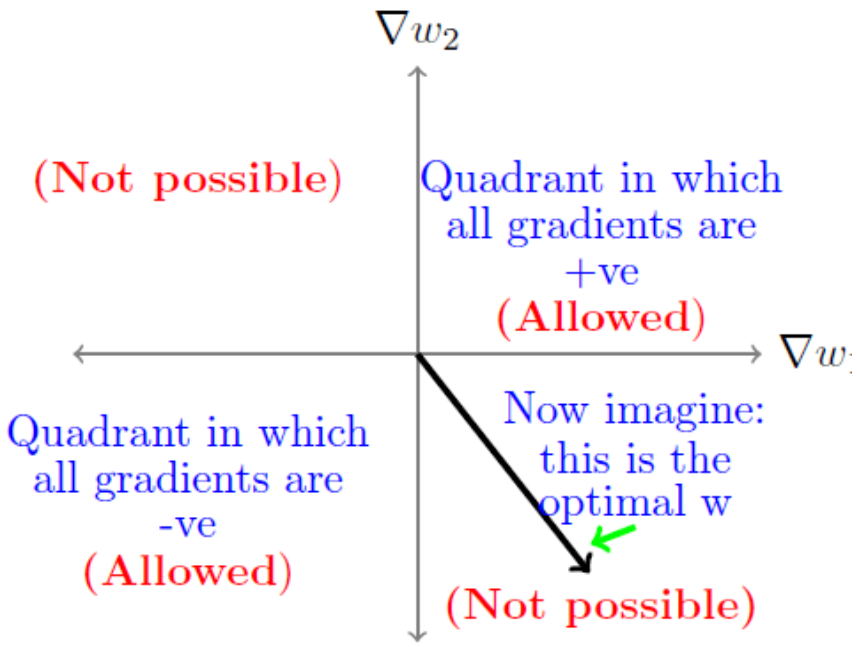
$$\nabla w_2 = \frac{\partial \mathscr{L}(\mathbf{w})}{\partial y} \frac{\partial y}{h_3} \frac{\partial h_3}{\partial a_3} \quad h_{22}$$

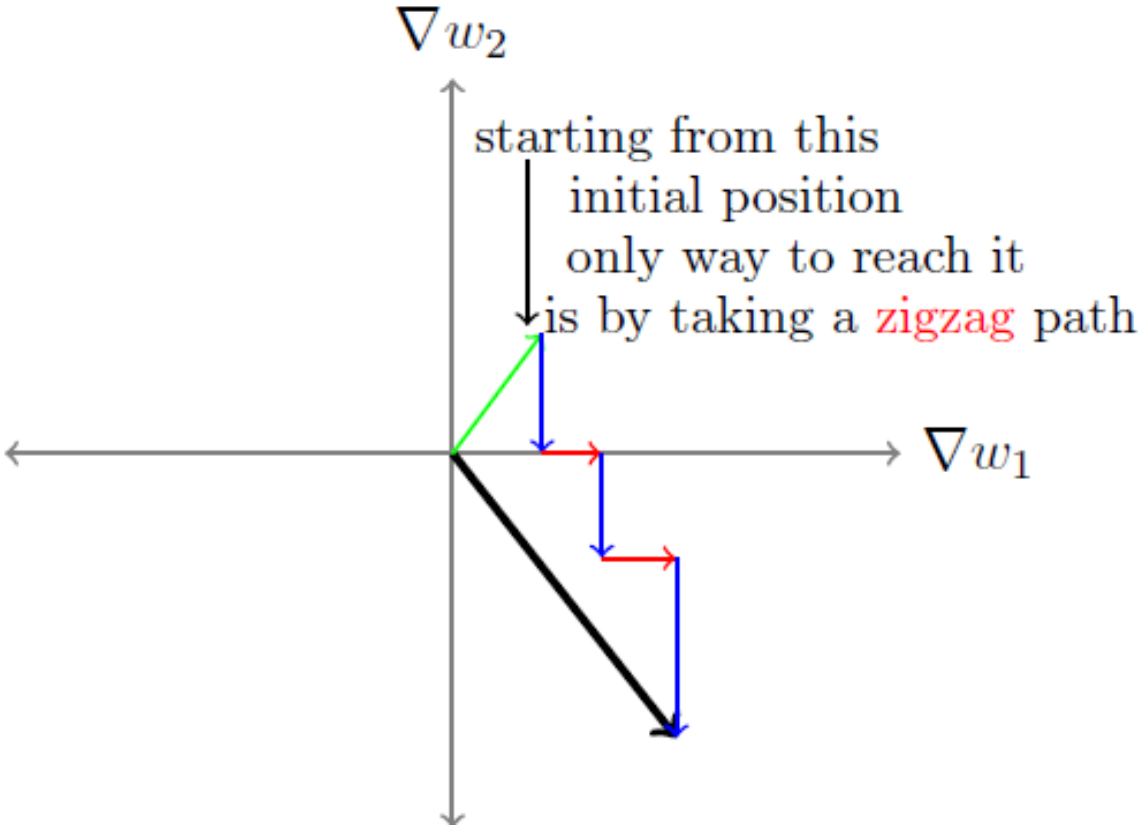


- Note that h_{21} and h_{22} are between $[0, 1]$ (*i.e.*, they are both positive)
- So if the first common term (in red) is positive (negative) then both ∇w_1 and ∇w_2 are positive (negative)
- Essentially, either all the gradients at a layer are positive or all the gradients at a layer are negative

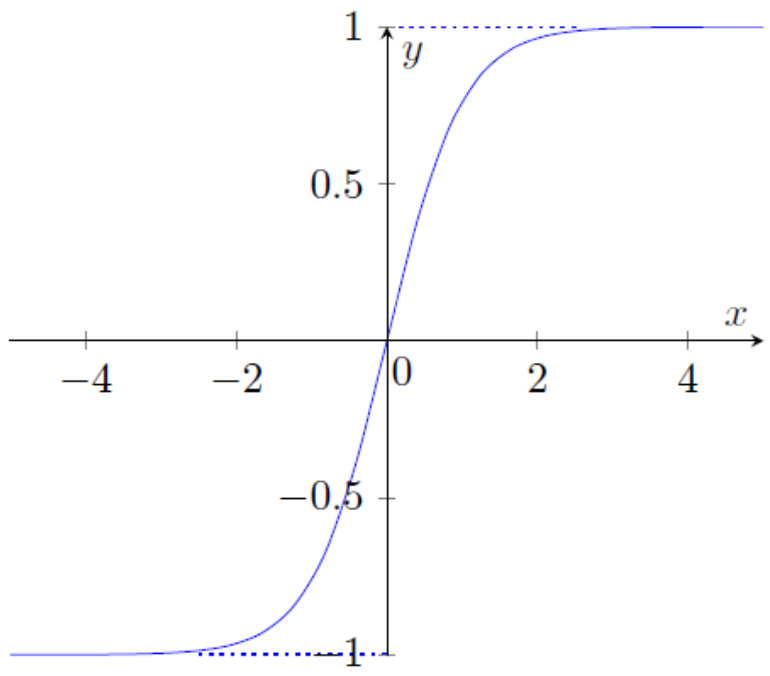
- Saturated neurons cause the gradient to vanish
- Sigmoids are not zero centered

- This restricts the possible update directions




∇w_2
starting from this
initial position
only way to reach it
is by taking a zigzag path
∇w_1

tanh(x)

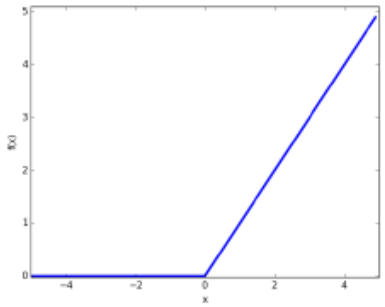


- Compresses all its inputs to the range [-1,1]
- Zero centered
- What is the derivative of this function?

$$\frac{\partial tanh(x)}{\partial x} = (1 - tanh^2(x))$$

- The gradient still vanishes at saturation
- Also computationally expensive

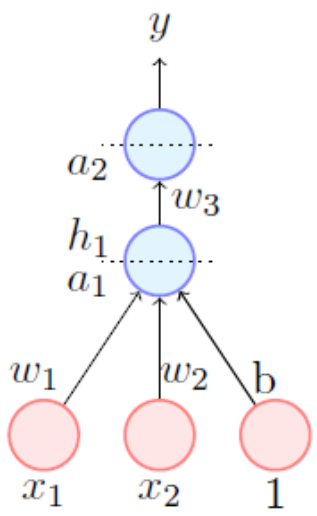
ReLU



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

Advantages of ReLU

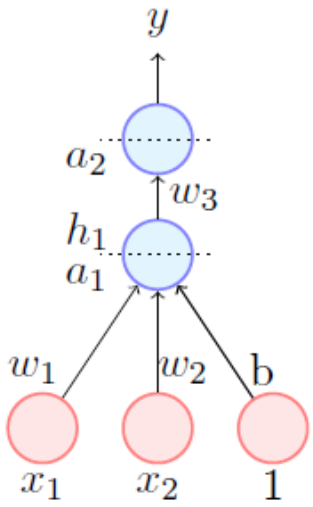
- Does not saturate in the positive region
- Computationally efficient
- In practice converges much faster than $sigmoid/tanh$[1]



- In practice there is a caveat
- Let's see what is the derivative of ReLU(x)

$$\frac{\partial ReLU(x)}{\partial x} = 0 \quad if \quad x < 0$$
$$= 1 \quad if \quad x > 0$$

- Now consider the given network
- What would happen if at some point a large gradient causes the bias b to be updated to a large negative value?

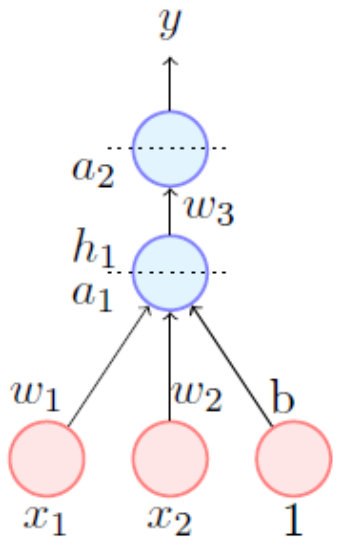


$$w_1x_1 + w_2x_2 + b < 0 \quad [if \quad b << 0]$$

- The neuron would output 0 [dead neuron]
- Not only would the output be 0 but during backpropagation even the gradient $\frac{\partial h_1}{\partial a_1}$ would be zero
- The weights w_1, w_2 and b will not get updated [$\because$ there will be a zero term in the chain rule]

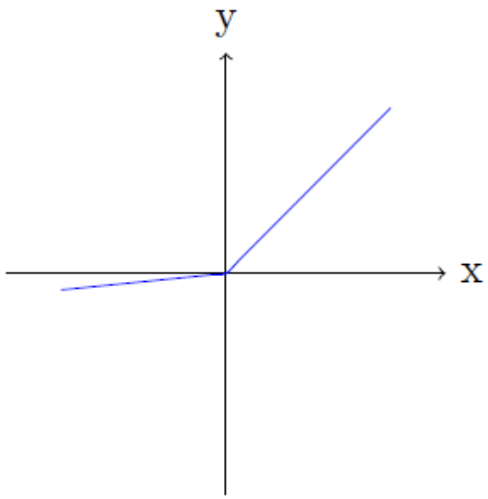
$$\nabla w_1 = \frac{\partial \mathscr{L}(\theta)}{\partial y} \cdot \frac{\partial y}{\partial a_2} \cdot \frac{\partial a_2}{\partial h_1} \cdot \frac{\partial h_1}{\partial a_1} \cdot \frac{\partial a_1}{\partial w_1}$$

- The neuron will now stay dead forever!!



- In practice a large fraction of ReLU units can die if the learning rate is set too high
- It is advised to initialize the bias to a positive value (0.01)
- Use other variants of ReLU (as we will soon see)

Leaky ReLU



f(x) = max(0.01x,x)

- No saturation
- Will not die ($0.01x$ ensures that at least a small gradient will flow through)
- Computationally efficient
- Close to zero centered ouputs

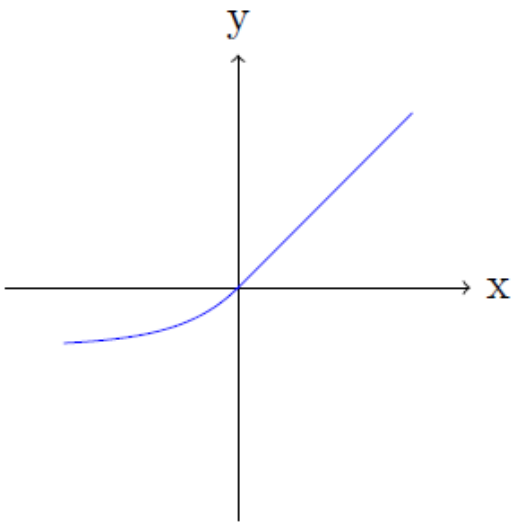
Parametric ReLU

$f(x) = \max(\alpha x, x)$

α *is a parameter of the model*

α *will get updated during backpropagation*

Exponential Linear Unit



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

- All benefits of ReLU
- $ae^x - 1$ ensures that at least a small gradient will flow through
- Close to zero centered outputs
- Expensive (requires computation of exp(x))

Maxout Neuron

$$max(w_1^T x + b_1, w_2^T x + b_2)$$

- Generalizes ReLU and Leaky ReLU
- No saturation! No death!
- Doubles the number of parameters

Things to Remember

- Sigmoids are bad
- ReLU is more or less the standard unit for Convolutional Neural Networks
- Can explore Leaky ReLU/Maxout/ELU

REGULARIZATION TECHNIQUES

- l_2 regularization
- Dataset augmentation
- Parameter Sharing and tying
- Adding Noise to the inputs
- Adding Noise to the outputs
- Early stopping
- Ensemble methods
- Dropout

- For l_2 regularization we have,

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

- For SGD (or its variants), we are interested in

$$\nabla\widetilde{\mathscr{L}}(w) = \nabla\mathscr{L}(w) + \alpha w$$

- Update rule:

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

- Requires a very small modification to the code

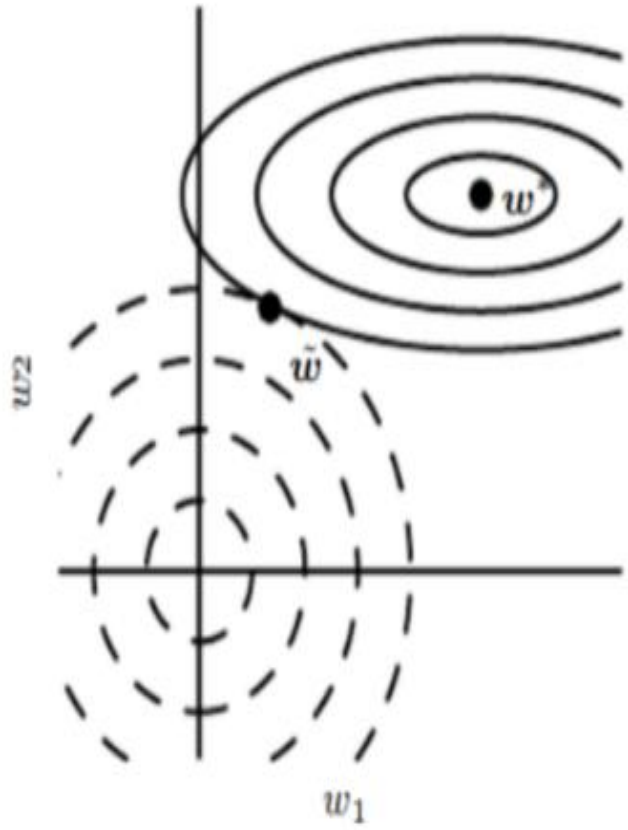
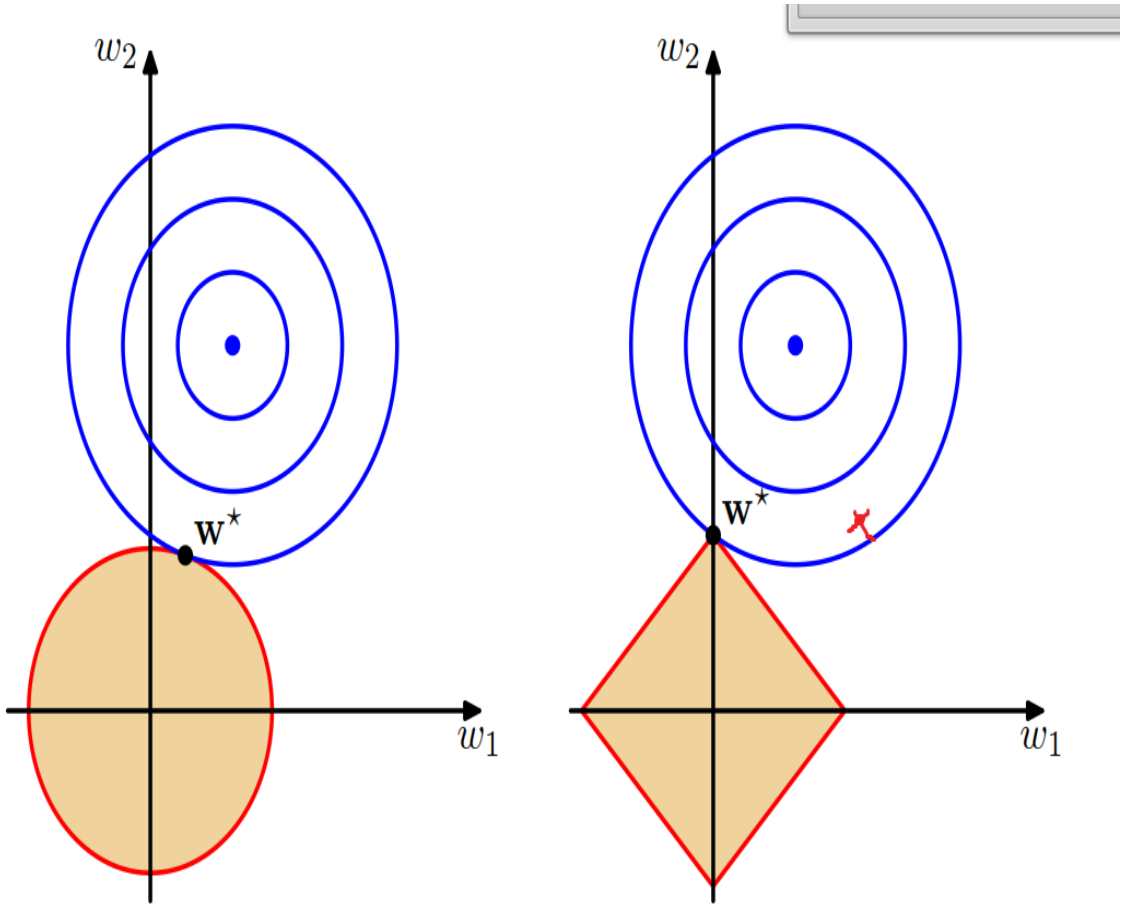
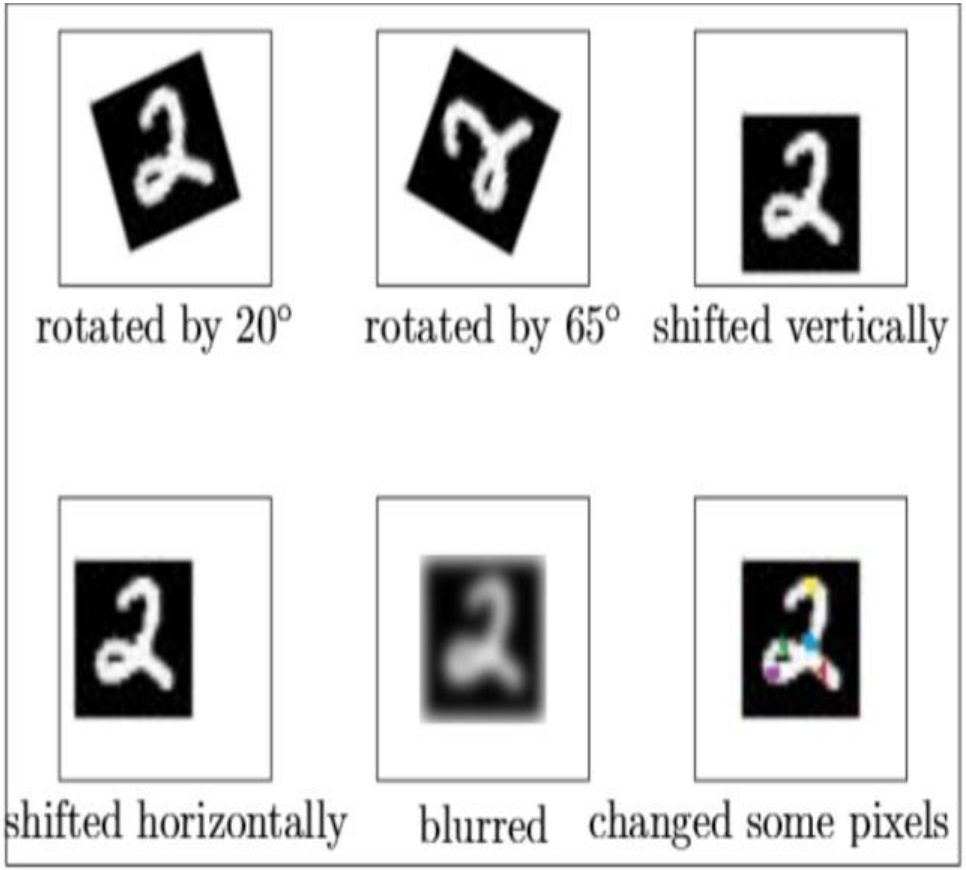

w^*
$\tilde{w}$
w_2
w_1

Figure 3.4 Plot of the contours of the unregularized error function (blue) along with the constraint region (3.30) for the quadratic regularizer $q = 2$ on the left and the lasso regularizer $q = 1$ on the right, in which the optimum value for the parameter vector $\mathbf{w}$ is denoted by $\mathbf{w}^\star$. The lasso gives a sparse solution in which $w_1^\star = 0$.

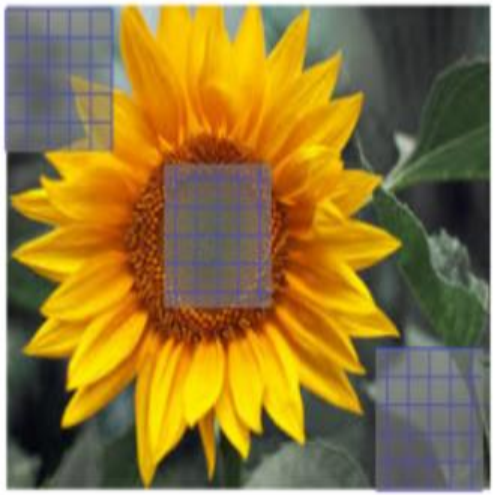


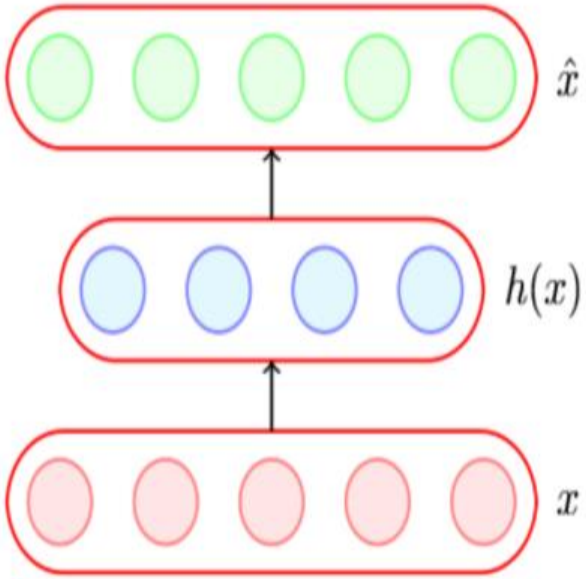
DATA AUGMENTATION

label = 2



- Typically, More data = better learning
- Works well for image classification / object recognition tasks
- Also shown to work well for speech





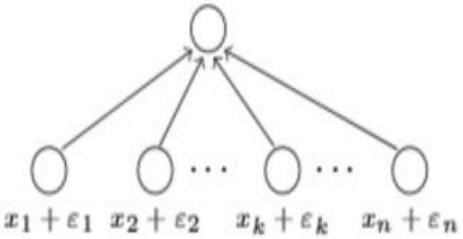
Parameter Sharing

- Used in CNNs
- Same filter applied at different positions of the image
- Or same weight matrix acts on different input neurons

Parameter Tying

- Typically used in autoencoders
- The encoder and decoder weights are tied.

GAUSSIAN NOISE ADDITION TO INPUT SAMPLES



$$\varepsilon \sim \mathcal{N}(0, \sigma^2)$$

We can show that for a simple input output neural network, adding Gaussian noise to the input is equivalent to weight decay (L_2 regularisation)

Can be viewed as data augmentation

$$\widetilde{x}_i = x_i + \varepsilon_i$$

$$\widehat{y} = \sum_{i=1}^{n} w_i x_i$$

$$\widetilde{y} = \sum_{i=1}^{n} w_i \widetilde{x}_i$$

$$= \sum_{i=1}^{n} w_i x_i + \sum_{i=1}^{n} w_i \varepsilon_i$$

$$= \widehat{y} + \sum_{i=1}^{n} w_i \varepsilon_i$$

We are interested in $E[(\tilde{y}-y)^2]$

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

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

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

$$= E\left[(\hat{y}-y)^2\right] + 0 + E\left[\sum_{i=1}^{n} w_i^2\varepsilon_i^2\right]$$

($\because \varepsilon_i$ is independent of ε_j and ε_i is independent of ($\hat{y}$-y))

$$= \left(E\left[(\hat{y}-y)^2\right] + \sigma^2\sum_{i=1}^{n} w_i^2\right) \quad \text{(same as } L_2 \text{ norm penalty)}$$

ADDING NOISE TO TARGETS / OUTPUTS

0	0	1	0	0	0	0	0	0	0

Hard targets

$$\text{minimize} : \sum_{i=0}^{9} p_i \log q_i$$

CROSS ENTROPY LOSS

$$\text{true distribution} : p = \{0, 0, 1, 0, 0, 0, 0, 0, 0, 0\}$$

$$\text{estimated distribution} : q$$

SOFTMAX FUNCTION TO MAP PREDICTED VALUES IN OUTPUT LAYER TO PROBABILITY

Intuition

- Do not trust the true labels, they may be noisy
- Instead, use soft targets

$\frac{\varepsilon}{9}$	$\frac{\varepsilon}{9}$	$1-\varepsilon$	$\frac{\varepsilon}{9}$	$\frac{\varepsilon}{9}$	$\frac{\varepsilon}{9}$	$\frac{\varepsilon}{9}$	$\frac{\varepsilon}{9}$	$\frac{\varepsilon}{9}$	$\frac{\varepsilon}{9}$

Soft targets

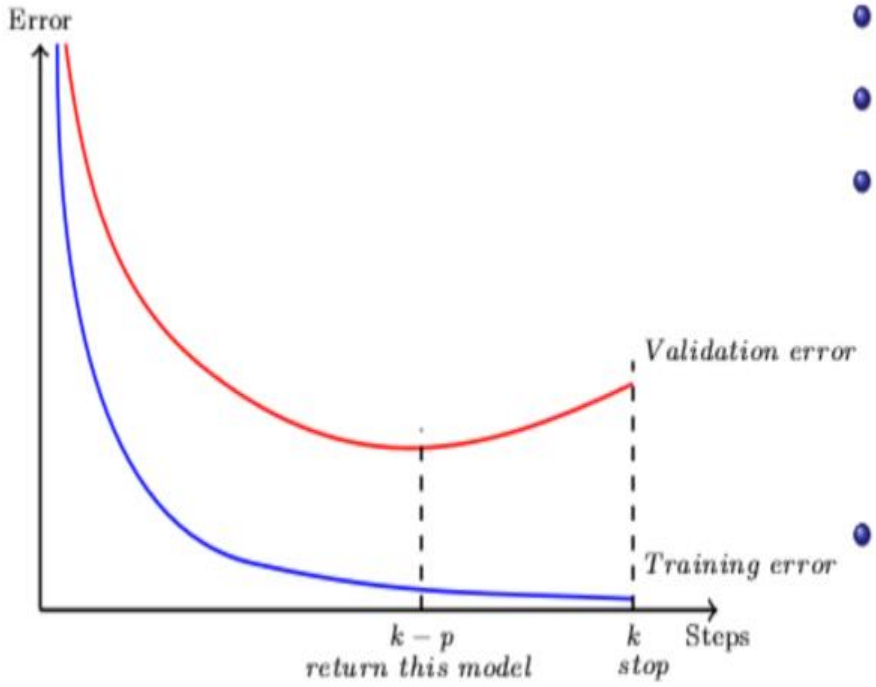
$$\varepsilon = \text{small positive constant}$$

$$\text{minimize} : \sum_{i=0}^{9} p_i \log q_i$$

$$\text{true distribution} + \text{noise} : p = \left\{ \frac{\varepsilon}{9}, \frac{\varepsilon}{9}, 1-\varepsilon, \frac{\varepsilon}{9}, \dots \right\}$$

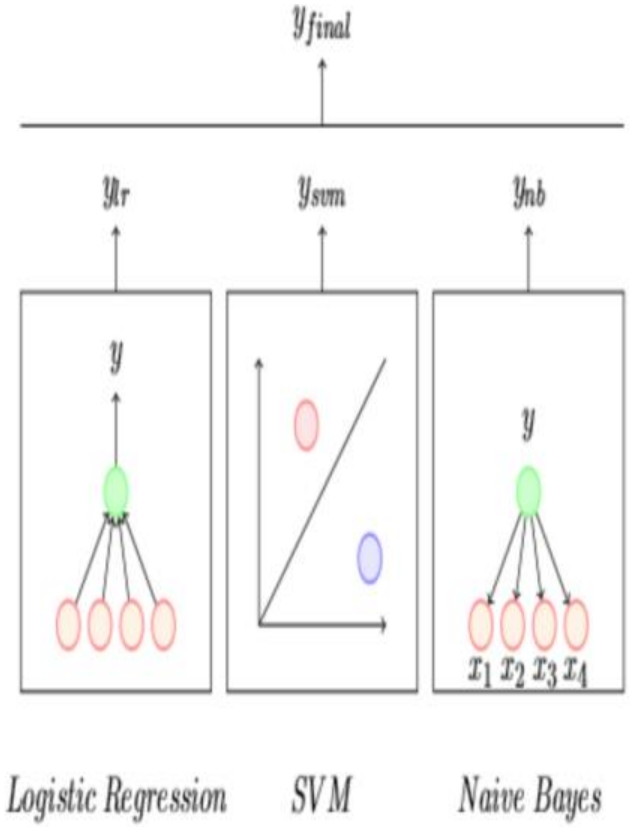
$$\text{estimated distribution} : q$$

EARLY STOPPING

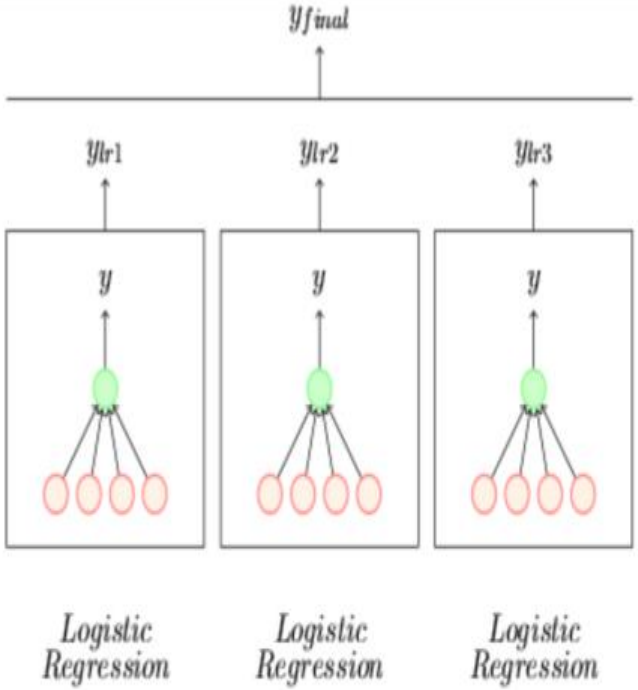


- Track the validation error
- Have a patience parameter p
- If you are at step k and there was no improvement in validation error in the previous p steps then stop training and return the model stored at step $k-p$
- Basically, stop the training early before it drives the training error to 0 and blows up the validation error

BAGGING



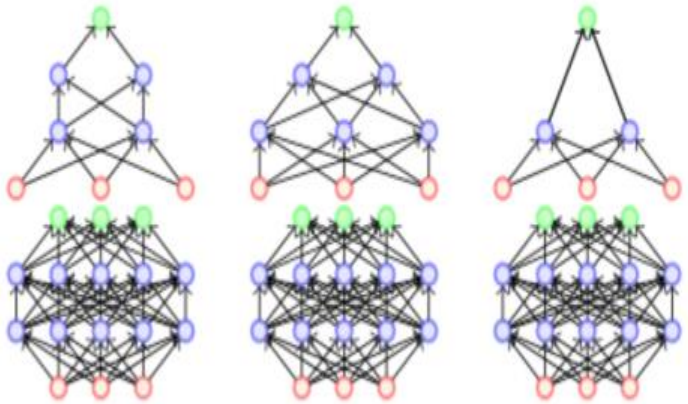
- Combine the output of different models to reduce generalization error
- The models can correspond to different classifiers
- It could be different instances of the same classifier trained with:
  - different hyperparameters
  - different features
  - different samples of the training data



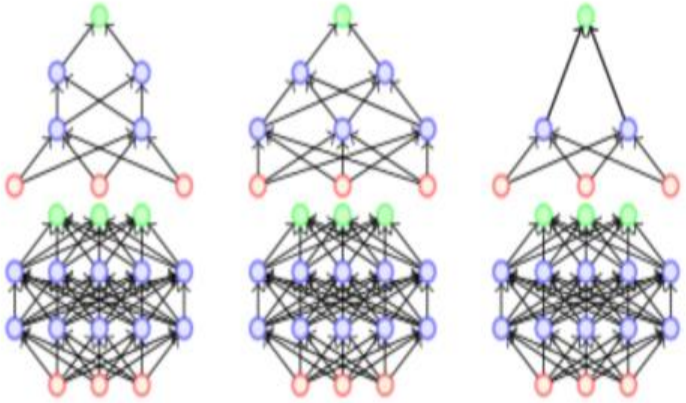
Each model trained with a different sample of the data (sampling with replacement)

- Bagging: form an ensemble using different instances of the same classifier
- From a given dataset, construct multiple training sets by sampling with replacement $(T_1, T_2, ..., T_k)$
- Train i^{th} instance of the classifier using training set T_i

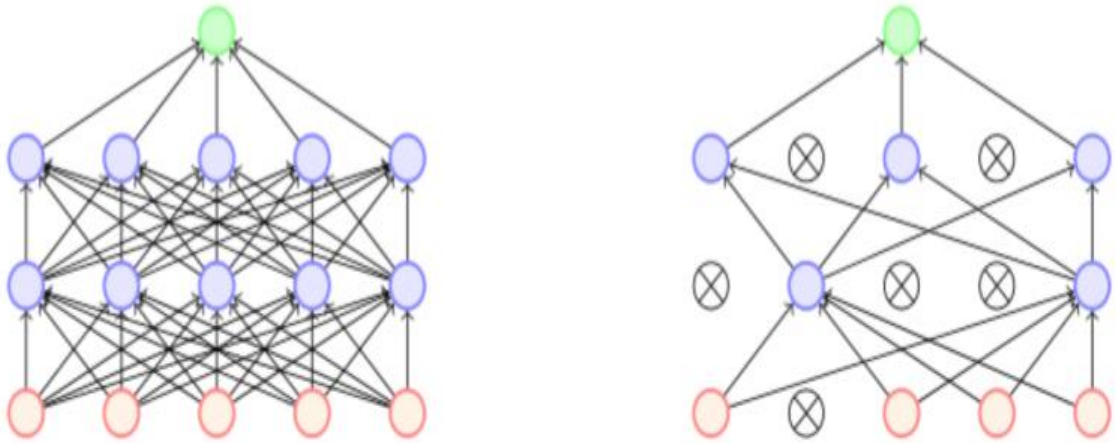
DROP OUT TECHNIQUE



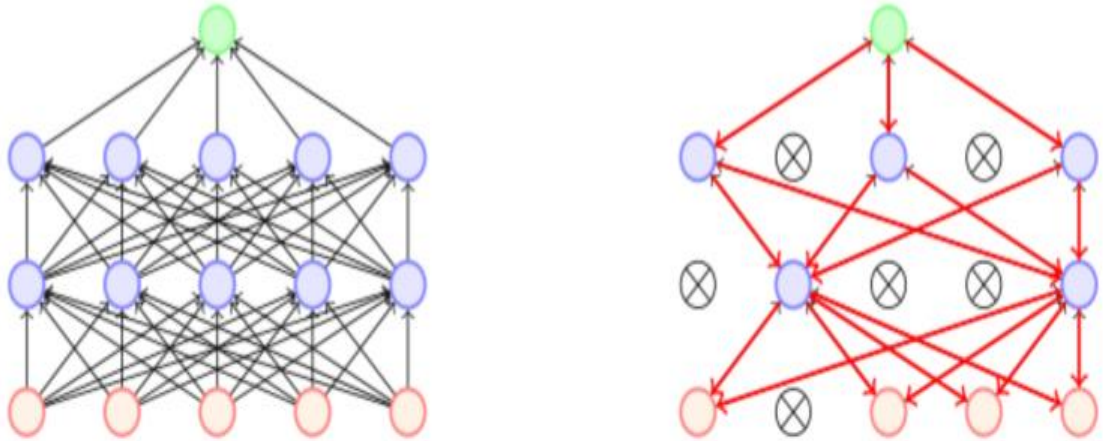
- Typically model averaging(bagging ensemble) always helps
- Training several large neural networks for making an ensemble is prohibitively expensive
- Option 1: Train several neural networks having different architectures(obviously expensive)
- Option 2: Train multiple instances of the same network using different training samples (again expensive)
- Even if we manage to train with option 1 or option 2, combining several models at test time is infeasible in real time applications



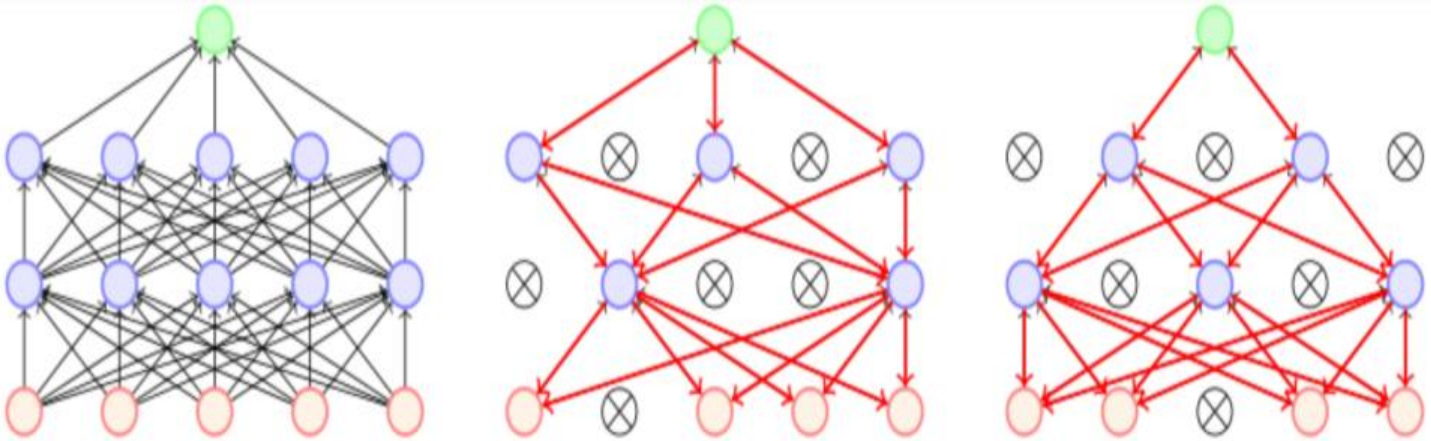
- Dropout is a technique which addresses both these issues.
- Effectively it allows training several neural networks without any significant computational overhead.
- Also gives an efficient approximate way of combining exponentially many different neural networks.



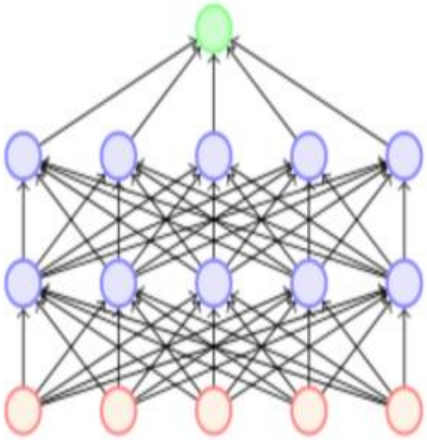
- Dropout refers to dropping out units
- Temporarily remove a node and all its incoming/outgoing connections resulting in a thinned network
- Each node is retained with a fixed probability (typically $p = 0.5$) for hidden nodes and $p = 0.8$ for visible nodes

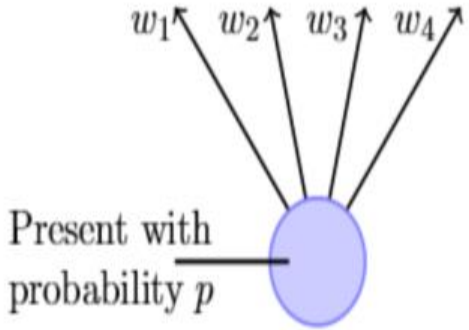


- We initialize all the parameters (weights) of the network and start training
- For the first training instance (or mini-batch), we apply dropout resulting in the thinned network
- We compute the loss and backpropagate
- Which parameters will we update? Only those which are active

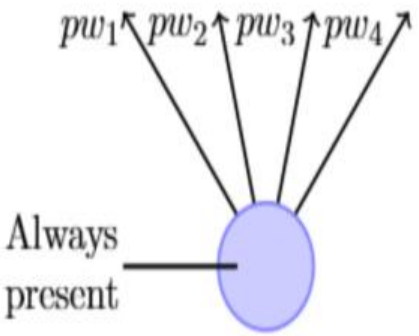


- For the second training instance (or mini-batch), we again apply dropout resulting in a different thinned network
- We again compute the loss and backpropagate to the active weights
- If the weight was active for both the training instances then it would have received two updates by now
- If the weight was active for only one of the training instances then it would have received only one updates by now
- Each thinned network gets trained rarely (or even never) but the parameter sharing ensures that no model has untrained or poorly trained parameters



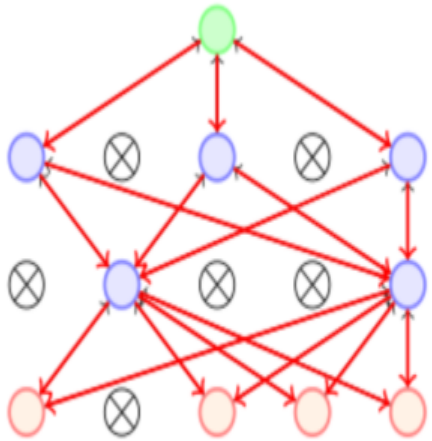


At training time

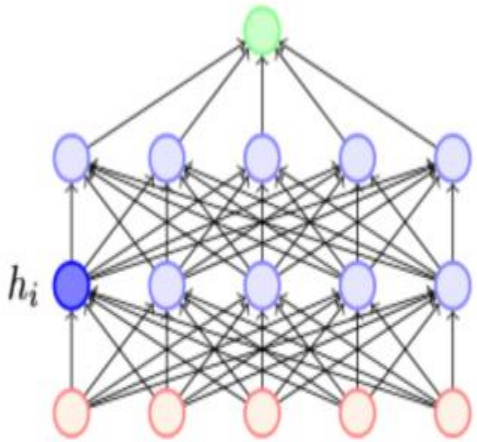


At test time

- What happens at test time?
- Impossible to aggregate the outputs of 2^n thinned networks
- Instead we use the full Neural Network and scale the output of each node by the fraction of times it was on during training



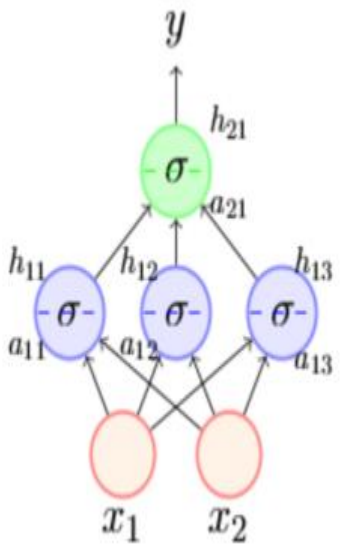
- Dropout essentially applies a masking noise to the hidden units
- Prevents hidden units from co-adapting
- Essentially a hidden unit cannot rely too much on other units as they may get dropped out any time
- Each hidden unit has to learn to be more robust to these random dropouts



- Here is an example of how dropout helps in ensuring redundancy and robustness
- Suppose h_i learns to detect a face by firing on detecting a nose
- Dropping h_i then corresponds to erasing the information that a nose exists
- The model should then learn another h_i which redundantly encodes the presence of a nose
- Or the model should learn to detect the face using other features

$$w_{11} = w_{21}$$

Assuming (scenario of equal weight initialization)



$$a_{11} = w_{11}x_1 + w_{12}x_2$$

$$a_{12} = w_{21}x_1 + w_{22}x_2$$

$$\therefore a_{11} = a_{12}$$

$$\therefore h_{11} = h_{12}$$

All neurons in layer 1 will get the same activation

Now what will happen during back propagation?

$$\nabla w_{11} = \frac{\partial \mathscr{L}(\mathrm{w})}{\partial y} \cdot \frac{\partial y}{\partial h_{11}} \cdot \frac{\partial h_{11}}{\partial a_{11}} \cdot x_1$$

$$\nabla w_{21} = \frac{\partial \mathscr{L}(\mathrm{w})}{\partial y} \cdot \frac{\partial y}{\partial h_{12}} \cdot \frac{\partial h_{12}}{\partial a_{12}} \cdot x_1$$

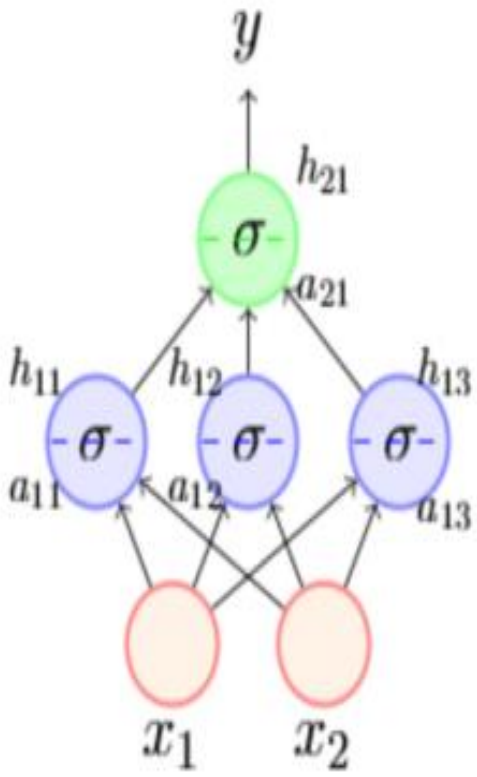
$$but \quad h_{11} = h_{12}$$

$$and \quad a_{12} = a_{12}$$

$$\therefore \nabla w_{11} = \nabla w_{21}$$

Let us try to initialize the weights to large random numbers

Most activations have saturated (vanishing gradient problem)



Let's try to initialize the weights to small random numbers

If all the activations in a layer are very close to 0 gradient of the weights will all be close to 0 (vanishing gradient problem)

Typically, we draw the weights from a unit Gaussian and scale them by $\frac{1}{\sqrt{n}}$

n = number of nodes in a layer.