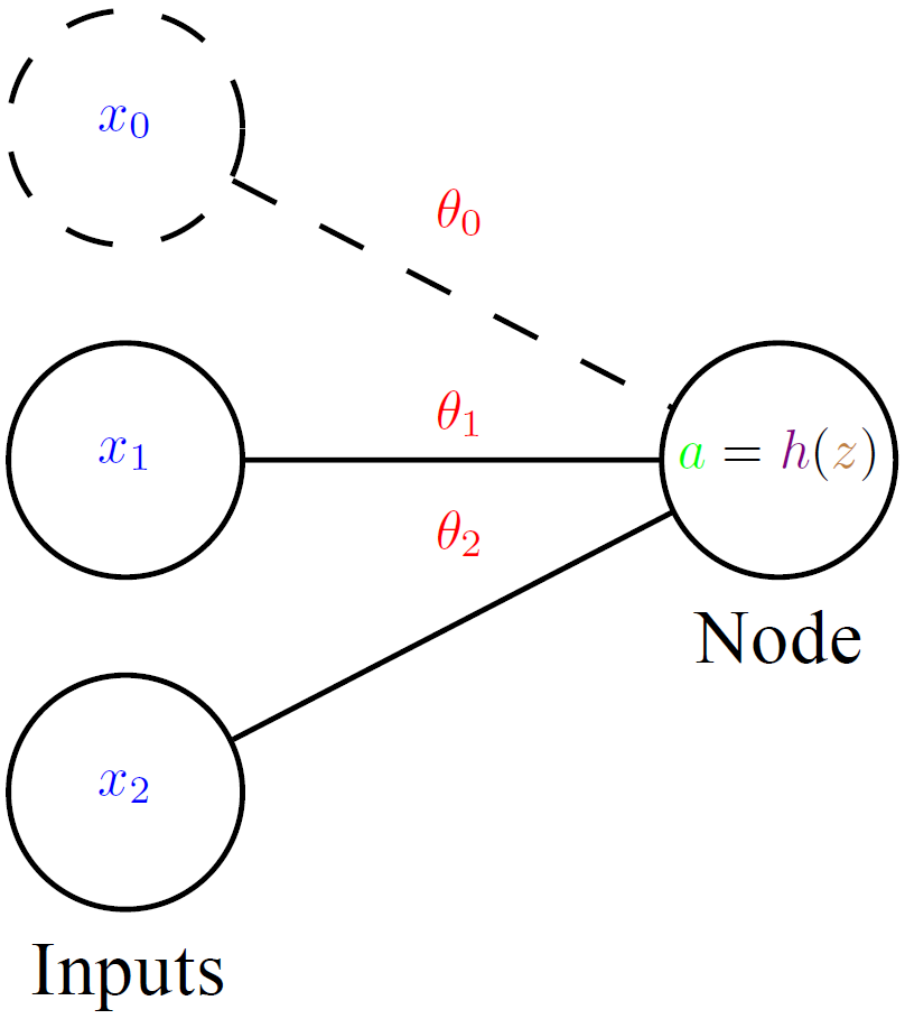
Artificial Neural Networks

Introduction

Notation and graphical representation

Single Node



- Inputs (data or outputs of other nodes):

$$x = \begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix}$$

- Weights (»regression coefficients«, need to be estimated)

$$\Theta = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \theta_3 \end{bmatrix}$$

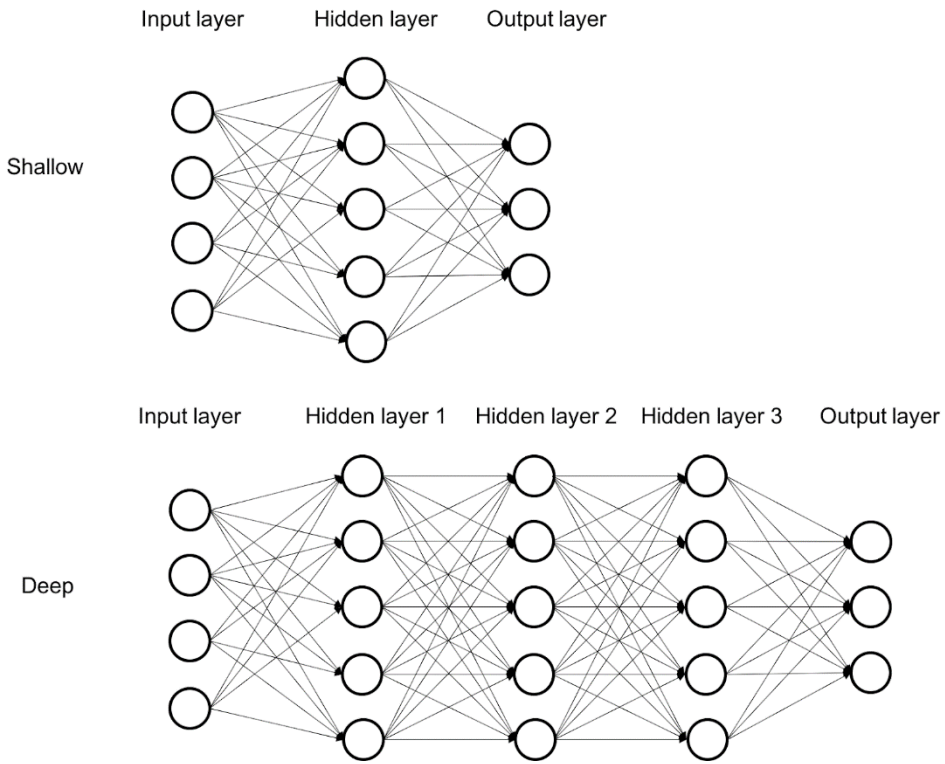
- Net input (“input function” = weighted sum)

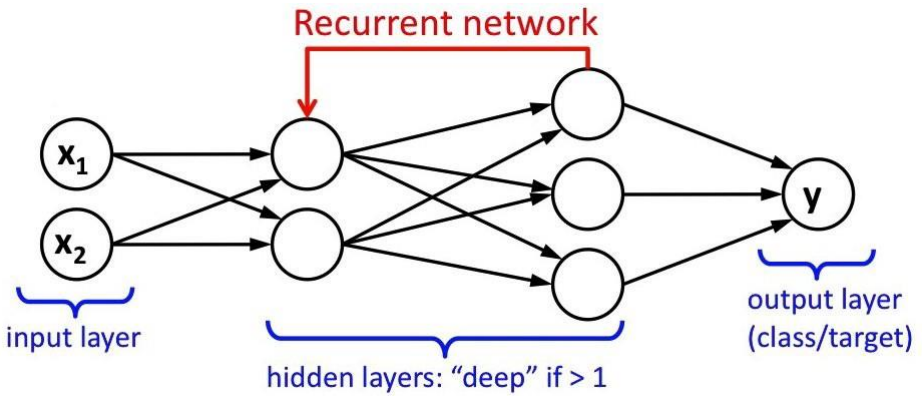
$$z = \theta' x = \theta_0 x_0 + \theta_1 x_1 + \theta_2 x_2$$

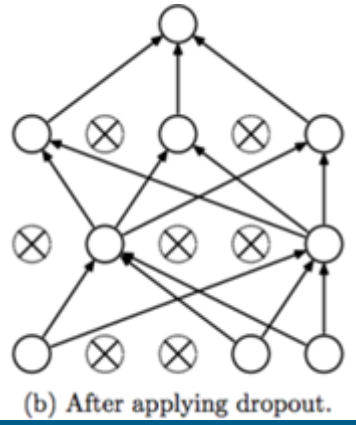
- Node’s output (transformed net input)

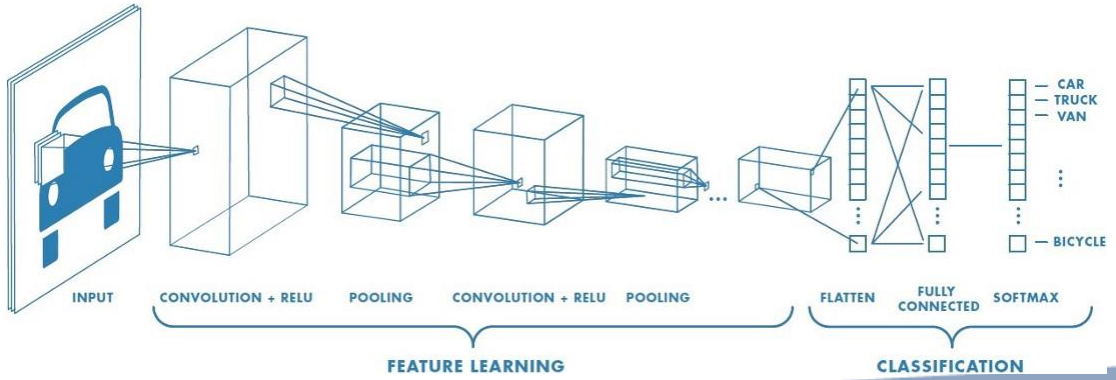
$a = h(z)$ $h()$ is "activation function"

Network(s)



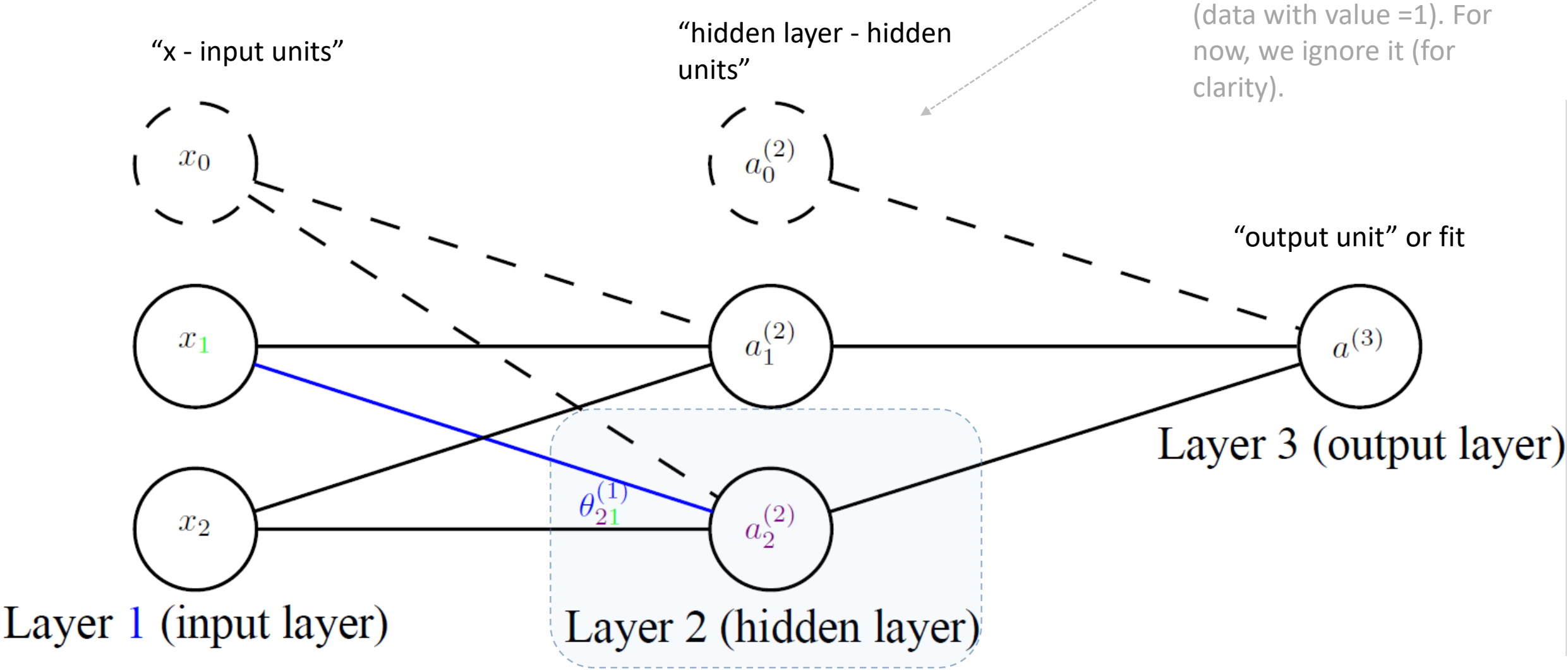



(b) After applying dropout.



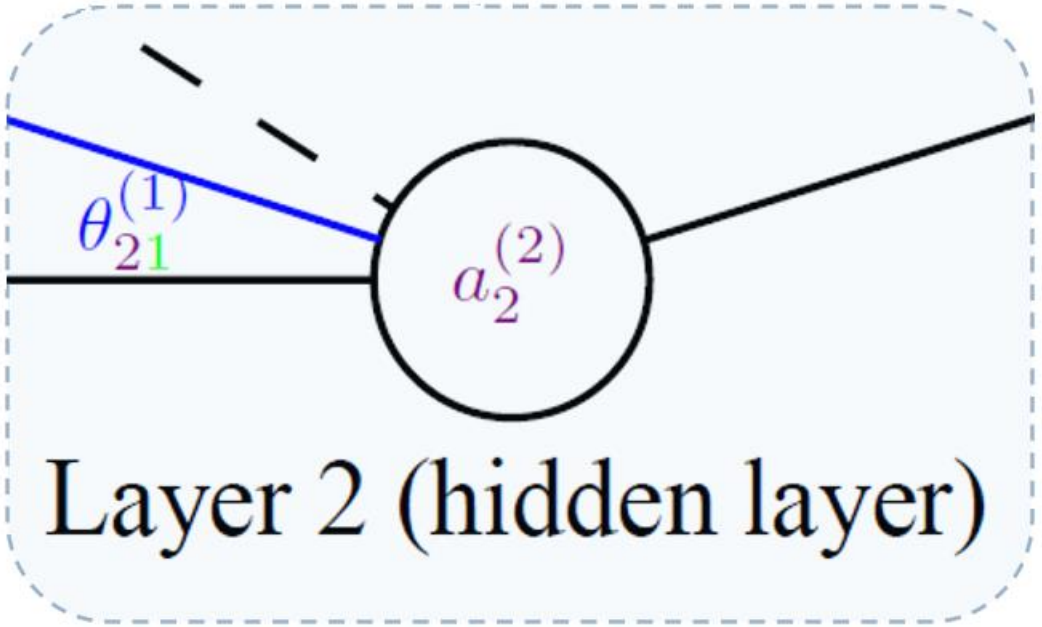
Network – 1 hidden layer

Dashed circle is called "bias term." Just like a constant in a regression (data with value =1). For now, we ignore it (for clarity).



Network – indexing convention

$a_2^{(2)}$ - 2nd activation unit/node in layer 2.
Brackets "(2)" denote layer. Subscript "2" denotes unit/node position.



$\theta_{21}^{(2)}$ - weight of x_1 in node 2 in layer 2.
Brackets "(2)" denote layer.
First subscript "2" denotes node position ($a_2^{(l)}$).
Second subscript "1" denotes previous node position ($x^{(1)}$ or $a_1^{(l)}$).

Estimation

Overview

- Steps
  1. Feed-Forward algorithm (delivers model fit)
  2. Back Propagation algorithm (delivers gradient of the “cost function” w.r.t. parameters)
  3. Gradient descent step (adjusts parameters)

- We iterate the above steps until a “stopping criteria” is met

1) Feed-Forward algorithm

- Process that delivers NN's prediction/fit.

Step 1: Get Input Layer (layer 1)

input layer is a layer where data enters

Step 2: Get Hidden Layer(s)

takes data from Layer 1 and performs transformations

Step 3: Output Layer

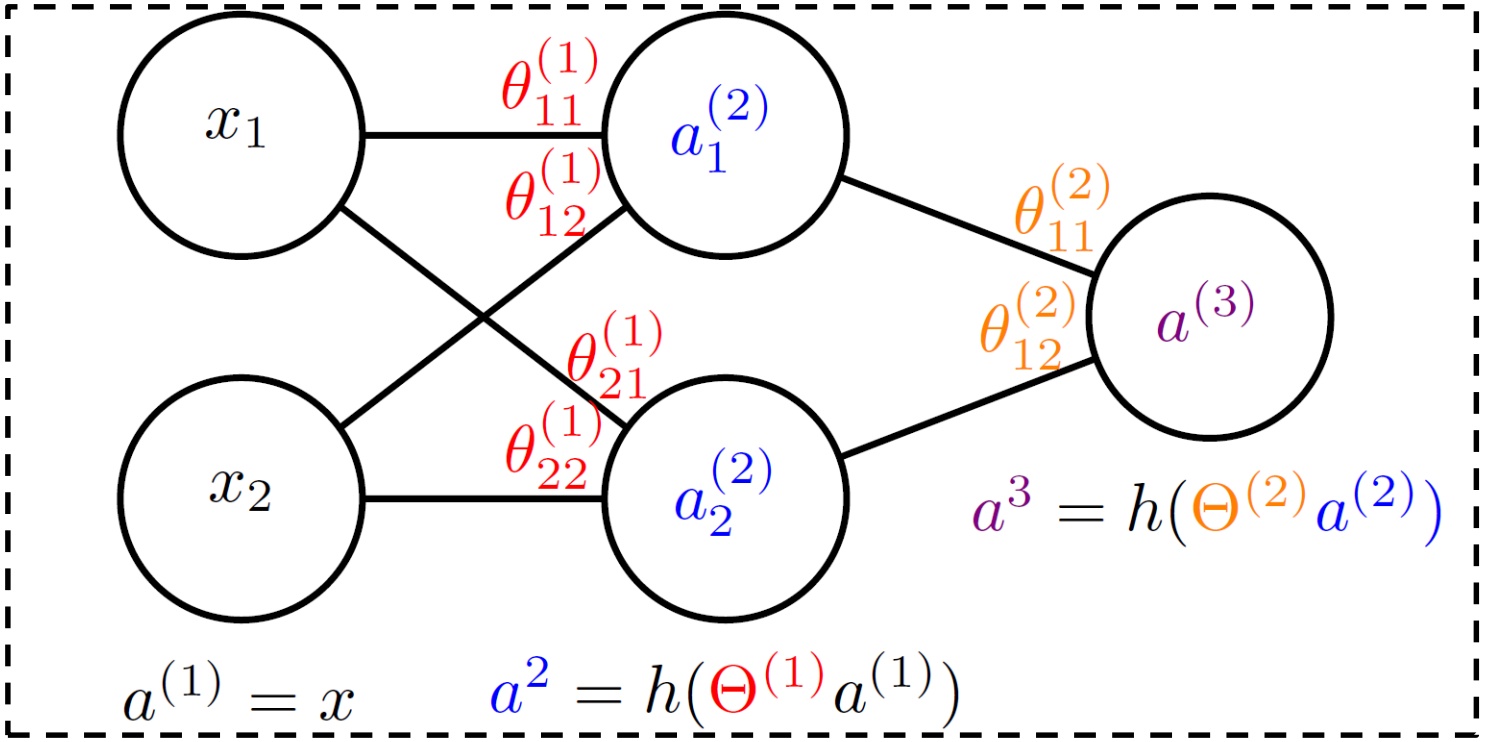
takes output of the final hidden layer and gives prediction

For clarity we ignore the "bias term" on the next slide.

Feed-Forward

Step 1 (input layer)

$$a^{(1)} = \begin{bmatrix} a_1^{(1)} \\ a_2^{(1)} \end{bmatrix} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$



Step 2 (eval. hidden layer)

$$a^{(2)} = \begin{bmatrix} a_1^{(2)} \\ a_2^{(2)} \end{bmatrix} = h\left(\Theta^{(1)} a^{(1)}\right) = h\left(\begin{bmatrix} \theta_{11}^{(1)} & \theta_{12}^{(1)} \\ \theta_{21}^{(1)} & \theta_{22}^{(1)} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}\right) = \begin{bmatrix} h(\theta_{11}^{(1)} x_1 + \theta_{12}^{(1)} x_2) \\ h(\theta_{21}^{(1)} x_1 + \theta_{22}^{(1)} x_2) \end{bmatrix}$$

Step 3 (eval. output layer)

$$a^{(3)} = a_1^3 = h\left(\Theta^{(2)} a^{(2)}\right) = h\left(\begin{bmatrix} \theta_{11}^{(2)} & \theta_{12}^{(2)} \end{bmatrix} \begin{bmatrix} a_1^{(2)} \\ a_2^{(2)} \end{bmatrix}\right) = h(\theta_{11}^{(2)} a_1^{(2)} + \theta_{12}^{(2)} a_2^{(2)})$$

For clarity, we omit time/unit index (e.g. x_{1t}, x_{1i}) and the bias units (e.g. $x_{0t} = 1$).

FeedForward - formulas

"$\mathrm{h}_\Theta(\mathrm{x})$" or "$a^{(L)}$" – NN output, model fit/prediction

Steps (for L-layers):

$$a^{(1)} = x$$
$$a^{(2)} = h\left(\Theta^{(1)} a^{(1)}\right)$$
$$\vdots$$
$$a^{(L)} = h\left(\Theta^{(L-1)} a^{(L-1)}\right) = h_\Theta(x)$$

Run for each training set (e.g. time period, data row)

If layer l has n_1 units and layer $l+1$ has n_2 units then $\Theta^{(l)} \in \mathbb{R}^{n_2 \times n_1}$. (assuming layer i includes bias)

$\Theta^{(\mathrm{l})}$, like regression model parameters, need to be estimated

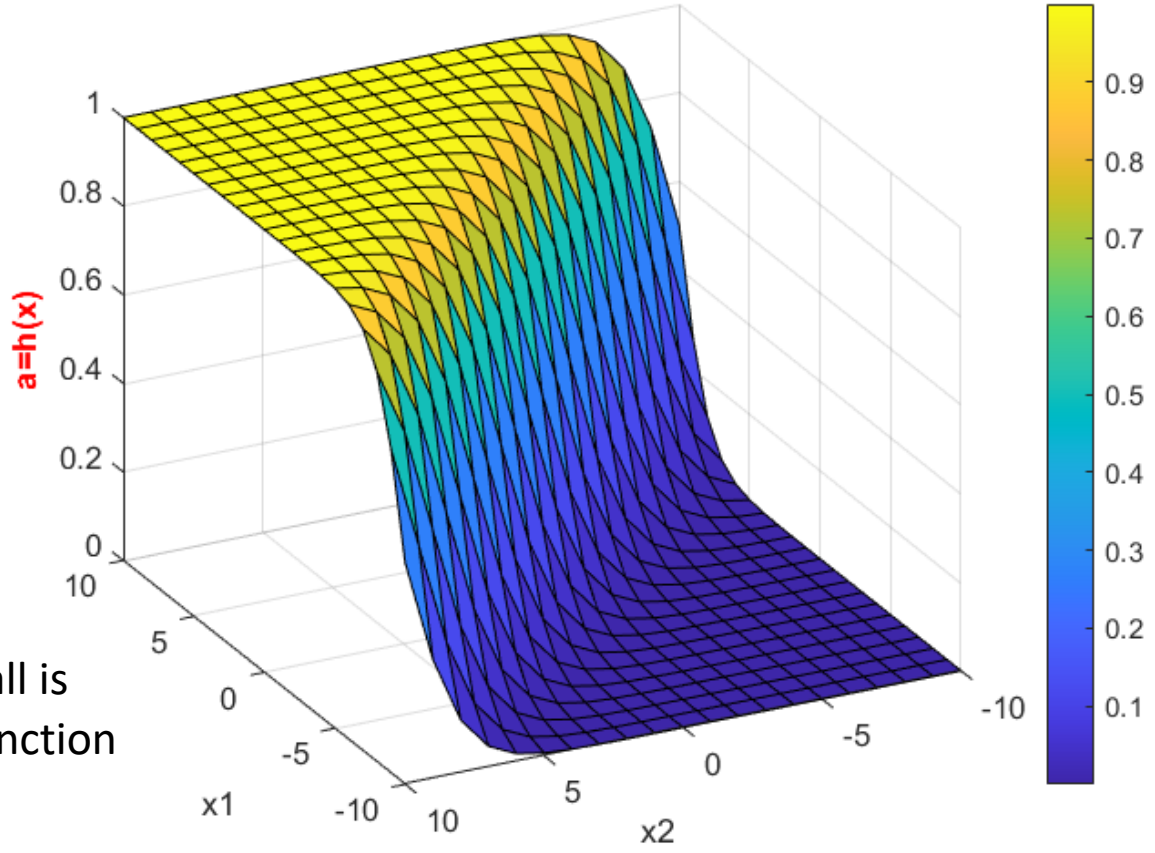
$\mathrm{h}(.)$ – is also called "activation function." It takes as an input "$\Theta^{(l)}\mathrm{a}^{(l)}$" (sometimes we write $\mathrm{z} = \Theta_i^{(l)}\mathrm{a}^{(l)}$)and outputs a scalar $\mathrm{a}^{(\mathrm{l}+1)}$. In practice, $\mathrm{h}(.)$ can take many forms (we will use sigmoid function) and can be different in each layer. But most $h(.)$ functions transform "$\Theta^{(i)}\mathrm{a}^{(i)}$" into a number between $[-1., 1]$ or $[0,1]$. This is why they are often called "squishing functions."

Sigmoid activation function (example)

$$a^{(i+1)} = h(z) = \frac{1}{1+e^{-z}};$$

$$z = \Theta^{(i)} a^{(i)}$$

$$= \theta_1^{(i)} x_1 + \theta_2^{(i)} x_2$$



Value of node $a^{i+1} \in (0,1)$ regardless of how big or small is $\{x_1, x_2\}$ or $\Theta^{(i)}$ (this example $\Theta^{(i)} = [1,1]$). Activation function squeezes $\Theta^{(i)} a^{(i)}$ into $(0,1)$ range.

(this helps prevent "exploding gradients" when estimating Θ)

Bias term

- For clarity we ignored the bias (constant) up till now.
- In practice we add bias (=a constant equal to 1, exact value is irrelevant) in all layers (input, hidden and output).
- Justification for bias is the same as in a regression setting: if you do not include bias and the “true” data-generating model includes the weights on the remaining coefficients will be biased (NN will not predict well).
- If bias is truly redundant for our task, the Θ coefficient associated with it will evaluate to 0.
- Without the bias a NN-layer can only approximate a limited set of functions, those that “go through the origin” (next slide)

Modification to the feed forward algorithm (remember to add bias when programming layers!):

Steps (for L-layers):

$a^{(1)} = x$ (add $a_0^{(1)} = 1$)

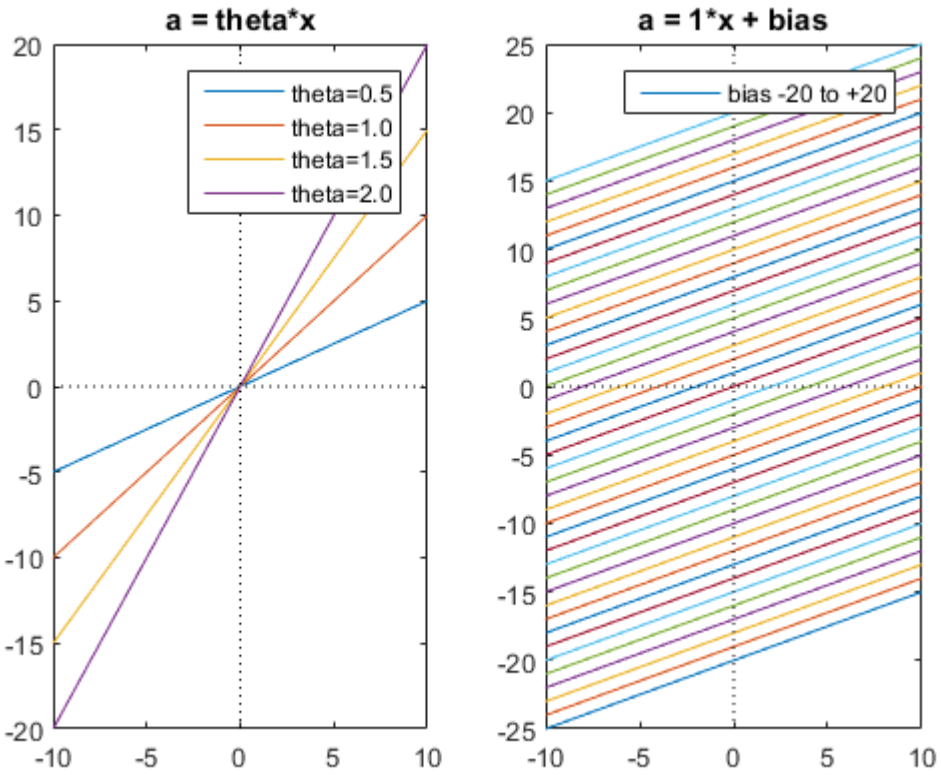
$a^{(2)} = h\left(\Theta^{(1)} a^{(1)}\right)$ (add $a_0^{(2)} = 1$)

$\vdots$ $\vdots$

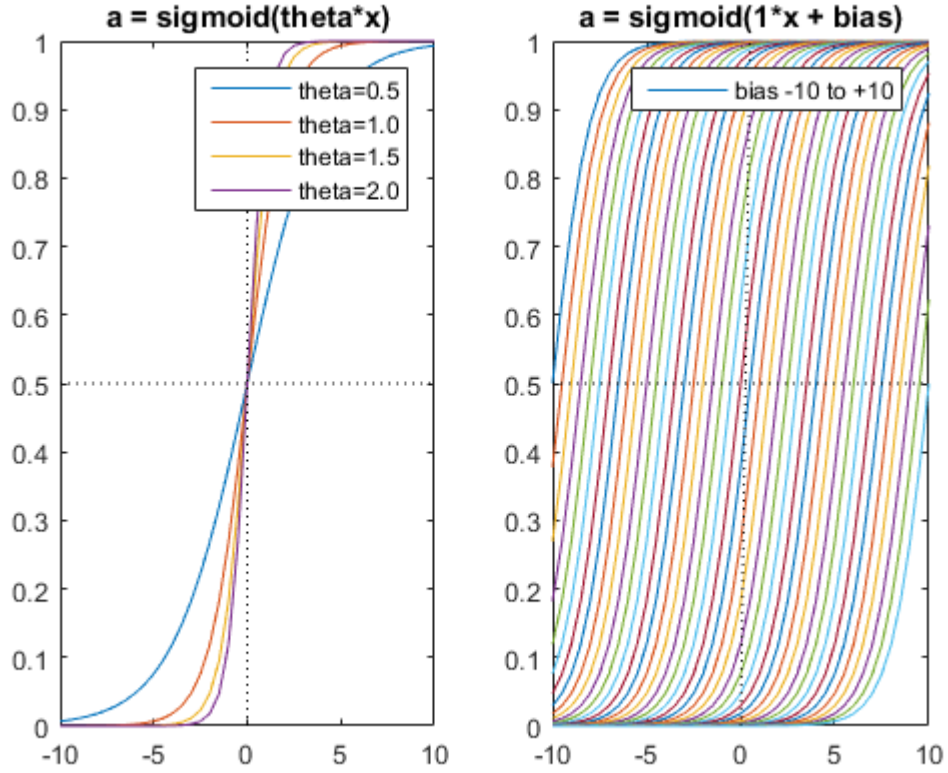
$a^{(L)} = h\left(\Theta^{(L-1)} a^{(L-1)}\right) = h_\Theta(x)$ (add $a_0^{(L)} = 1$)

Bias expands the range of functions that a layer can mimic

Identity function activation



Sigmoid activation



Think about how the NN changes theta if we estimate a no-bias layer and the true process includes a bias?

2) Backpropagation algorithm

- Feed-forward calculates model's fit/prediction.
- Back-Propagation facilitates "learning" (= Θ parameter estimation)
- NN parameters are typically optimized with (stochastic) gradient descent algorithm, requires gradient of the cost function ($J(\Theta)$; on next slide) w.r.t parameters: $\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta)$
- In principle, one could derive gradient analytically
- Back-Propagation is an algorithm which calculates gradient with little computational effort

Cost Function $J(\Theta)$

$sometimes\ written\ as\colon\ a^{(L)}(x_i)\ or\ h(x_i)$

- We estimate Θ by minimizing some cost function
- In regression context the cost function is typically the mean squared error

$$\mathrm{J}(\Theta) = \frac{1}{2m}\sum_{i=1}^{m}\left(a^{(L)} - y_i\right)^2$$

- "Single-run" cost is usually called Loss function $L(\Theta)$

$$L(\Theta) = \frac{1}{2}\left(a^{(L)} - y_i\right)^2$$

- example for 3-layer NN:

$$a^{(3)} = h_3\left(\Theta^{(2)} h_2\left(\Theta^{(1)} x_i\right)\right)$$

$a^{(1)}$

$a^{(2)}$

For regression models (as opposed to classification), the last activation function (h_3) is often just an identity function.

3) Gradient Descent

$L(.)$ is:
Classification
Binary classification-> logistic loss function
Multiple classification -> categorical cross-entropy loss
Regression
MSE, MAE, Huber loss,…

- Parameters (e.g.): $\Theta^{(1)}$, $\Theta^{(2)}$
- Cost Function: $J(\Theta) = \frac{1}{2m}\sum_{i=1}^{m} L\left(a^{(L)}, y_i\right)^2$ ($a^{(2)}$)

- Gradient Descent (repeat):
  - Initialize Θ
  - Compute partial derivatives (Back Propagation): $\Delta^{(l)} = \frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta)$
  - Update parameters: $\Theta_{ij}^{(l)} = \Theta_{ij}^{(l)} - \alpha d\Theta_{ij}^{(l)}$

$\alpha \in (0,1)$ – the learning rate. A small positive value, it needs to be tuned in.

Gradient of Cost Function in layer (l): $\Delta^{(l)}$

- Gradient $\Delta^{(l)}$ is a matrix with partial derivatives of $J(\Theta)$ w.r.t $\Theta^{(l)}$.
- We calculate it with Back-Propagation algorithm:

  Steps:

  Forward Propagation: $\{a^{(1)}, a^{(2)}, \ldots, a^{(L)}\}$

  Compute last layer error: $\delta^{L} = a^{(L)} - y_i$

  Propagate the error backwards: $\delta^{(l)} = \left(\Theta^{(l)}\right)^{\top} \delta^{(l+1)} .* h'(z) \qquad l = L - 1 \ldots 1$

- We next look at Back Propagation in more detail.

Forward Propagation and Back Propagation

$$a^{(1)} = x \rightarrow z^{(1)} = \Theta^{(1)} a^{(1)} \rightarrow a^{(2)} = h\left(z^{(1)}\right) \rightarrow z^{(2)} = \Theta^{(2)} a^{(2)} \rightarrow z^{(2)} = \Theta^{(2)} a^{(2)} \rightarrow a^{(3)} = h\left(z^{(2)}\right) \rightarrow L(y, a^{(3)})$$

- We do forward propagation to get $\{a^{(1)}, \dots, a^{(3)}\}$ and $L(y, a^{(3)})$
- We do back propagation to get $\Delta = \frac{\partial}{\partial \Theta} C(\Theta)$, partial derivatives of the cost function w.r.t. parameters
- Cost function is simply the sum of losses $C(\Theta) = \frac{1}{m} \sum_{i=1}^{m} L(y^{(i)}, a^{(3)})$
- Back propagation is an algorithm that iteratively calculates partial derivatives of the loss function wrt. parameters ($\frac{\partial}{\partial \Theta} L(\Theta)$) (for i=1,..,m), accumulates them (=sums over iterations) and returns $\Delta = \frac{\partial}{\partial \Theta} C(\Theta)$.
- Δ is just a sum of $\frac{\partial}{\partial \Theta} L(\Theta)$ (over 1,…,m).
- To understand back propagation, we next have look at how the algorithm calculates a single $\frac{\partial}{\partial \Theta} L(\Theta)$ for a single hidden-layer network.

Reminders

- Chain rule $\frac{\partial f(y(x))}{\partial x} = \frac{\partial f}{\partial y}\frac{\partial y}{\partial x}$
- Derivative of sigmoid function: $\frac{\partial h(x)}{\partial x} = h(x)(1 - h(x))$

Back propagation – calculation of Δ

$\Delta^{(1)}$ $\Delta^{(2)}$

$$a^{(1)} = x \rightarrow z^{(1)} = \Theta^{(1)\top} a^{(1)} \rightarrow a^{(2)} = h\left(z^{(1)}\right) \rightarrow z^{(2)} = \Theta^{(2)\top} a^{(2)} \rightarrow a^{(3)} = h\left(z^{(2)}\right) \rightarrow L(y, a^{(3)})$$

- $\Delta^{(2)} = \frac{\partial L}{\partial \Theta^{(2)}} = \frac{\partial L}{\partial a^{(3)}} \frac{\partial a^{(3)}}{\partial z^{(2)}} \frac{\partial z^{(2)}}{\partial \Theta^{(2)}} = \delta^{(2)} a^{(2)\top}$

$$\delta^{(2)} = \frac{\partial L}{\partial z^{(2)}} = \frac{\partial L}{\partial a^{(3)}} \frac{\partial a^{(3)}}{\partial z^{(2)}} = \frac{\partial L}{\partial a^{(3)}} .* h'\left(z^{(2)}\right)$$

$$\frac{\partial z^{(2)}}{\partial \Theta^{(2)}} = d\Theta^{(2)} = a^{(2)\top}$$

- $\Delta^{(1)} = \frac{\partial L}{\partial \Theta^{(1)}} = \frac{\partial L}{\partial a^{(3)}} \frac{\partial a^{(3)}}{\partial z^{(2)}} \frac{\partial z^{(2)}}{\partial a^{(2)}} \frac{\partial a^{(2)}}{\partial z^{(1)}} \frac{\partial z^{(1)}}{\partial \Theta^{(1)}} = \delta^{(1)} a^{(1)\top}$

$$\delta^{(1)} = \frac{\partial L}{\partial z^{(1)}} = \frac{\partial L}{\partial a^{(3)}} \frac{\partial a^{(3)}}{\partial z^{(2)}} \frac{\partial z^{(2)}}{\partial a^{(2)}} \frac{\partial a^{(2)}}{\partial z^{(1)}} = \Theta^{(2)\top} \delta^{(2)} .* h'(z^{(1)})$$

$$\frac{\partial z^{(1)}}{\partial \Theta^{(1)}} = d\Theta^{(1)} = a^{(1)\top}$$

BackProp gives you $\delta^{(l)}$, the "error" (vector) in layer *l*.

Back Propagation Algorithm

Set $\Delta^{(l)} = 0 \quad \forall\, l$

for each $\{x_i, y_i\}$:

(error and "gradient in the last layer)

$\delta^{(L)} = \frac{\partial L}{\partial a^{(L+1)}} .* h'\left(z^{(L)}\right)$

$\Delta^{(L)} = \Delta^{(L)} + \delta^{(L)} a^{(L)\top} \ (accumulate)$

(prop. error and gradient to other layers)

$\delta^{(l)} = \Theta^{(L+1)\top} \delta^{(l+1)} .* h'\left(z^{(l)}\right)$

$\Delta^{(l)} = \Delta^{(l)} + \delta^{(l)} a^{(l)\top} \ (accumulate)$

Exact form of $\delta^{(l)}$ depend on the loss function $L(y, a^{(L)})$ and "squishing" functions ($h()$).

- Regression, L=MSE, $h^{(L)}$=identity, and $h^{(l)}$ =sigmoid:

$$L\left(y, a^{(L)}\right) = \frac{1}{2}\left(y - a^{(L)}\right)^2$$

(can be shown) $\delta^{(L)} = a^{(L)} - y$

- Regression, L=MSE, $h^{(L)}$=sigmoid, and $h^{(l)}$ =sigmoid:

$$L\left(y, a^{(L)}\right) = \frac{1}{2}\left(y - a^{(L)}\right)^2$$

(not as nice) $\delta^{(L)} = (a^{(L)} - y)(1 - a^{(L)})a^{(L)}$

- Binary classification, L=logistic and h=sigmoid:

$$L\left(y, a^{(3)}\right) = -(y \log\left(a^{(3)}\right) + (1 - y) log(1 - a^{(3)})$$

(can be shown) $\delta^{(L)} = a^{(L)} - y$

Estimation – all steps together

Set $\{\Theta^{(l)} = rnd.\ on\ 1^{st}\ run, \Delta^{(l)} = 0\} \quad \forall\ l$

Loop over $\{x_i, y_i\}$:

(1) Forward Propagation

$a^{(1)} = x_i$ (+add bias to each $a^{(l)}$)

$a^{(2)} = h\left(\Theta^{(1)} a^{(1)}\right)$

$\vdots$

$a^{(L)} = h\left(\Theta^{(L-1)} a^{(L-1)}\right) = h_\Theta(x)$

(2) Backward Propagation

$\delta^{(L)} = \frac{\partial L}{\partial a^{(L+1)}} .* h'\left(z^{(L)}\right)$ — last layer

$\Delta^{(L)} = \Delta^{(L)} + \delta^{(L)} a^{(L)\mathsf{T}}$

$\delta^{(l)} = \Theta^{(L+1)\mathsf{T}} \delta^{(l+1)} .* h'\left(z^{(l)}\right)$ — other layers

$\Delta^{(l)} = \Delta^{(l)} + \delta^{(l)} a^{(l)\mathsf{T}}$

(except for bias!) gradient for bias is:

$\Delta^{(L)} = \Delta^{(L)} + \delta^{(L)}$

Loop ends here

(3) Perform Gradient Descent

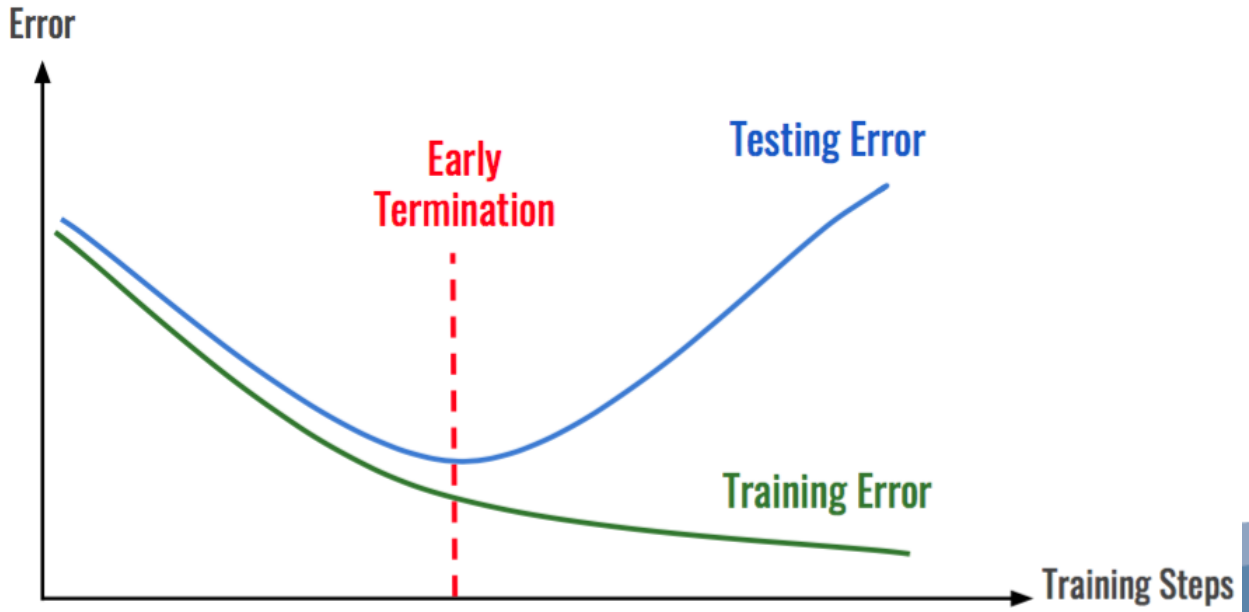
$$D^{(l)} = \frac{1}{m} \Delta^{(l)}$$

$$\Theta^{(l)} = \Theta^{(l)} - \alpha D^{(l)}$$

(4) Repeat (1)-(3) till stopping criteria is met

Example of stopping criteria

- Split data to train and test set (say 80-20%)
- For each “epoch” (epoch=training cycle/training step)
  - Estimate Θ on train set (80%)
  - Calculate testing error ($\mathrm{MSE}(\Theta)$) on test set (20%). Stop when it starts to increase.



Further topics

Further Topics: Initialization of weights and bias

- Never set all weights to 0. If every neuron computes the same output and ther weights are the same then they will also compute the same gradient during back propagation, have the same parameter update,... There is no source of asymetry between the neurons.
- Typically a small random number ϵ for each parameter (ok for small networks).
- Sometimes the neurons weight vector is scaled by the number of input neurons leading to it: $\epsilon/\sqrt{n_{in}}$
- For sigmoid & tanh (on next slide) Xavier initialization is often used:

$$W \sim U\left[-\sqrt{\frac{6}{n^{(l)}+n^{(l+1)}}}, \sqrt{\frac{6}{n^{(l)}+n^{(l+1)}}}\right]$$

Where $n^{(l)}$ is the number of input units to W (fan-in)
and $n^{(l+1)}$ is the number of output units from W (fan-out)

Further topics: Activation functions

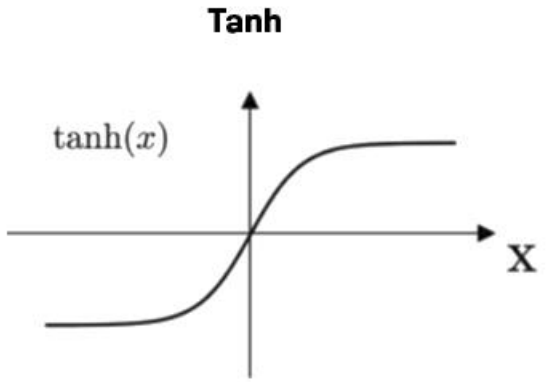
- Sum of linear functions is a linear function itself. Therefore, NN without nonlinear activation functions is just a linear regression model – it cannot model nonlinearities.
- Activation functions enable us to model nonlinearities.

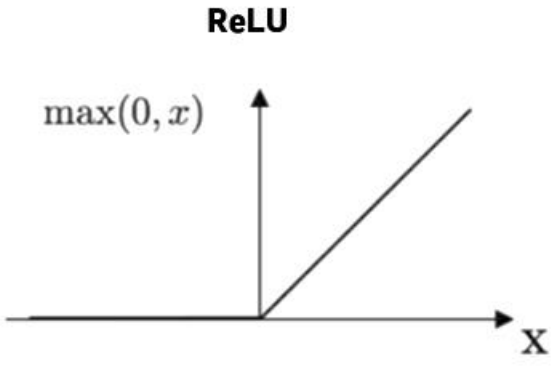
In practice, ReLu is a good choice to start with. From here the «trial&error» process starts.

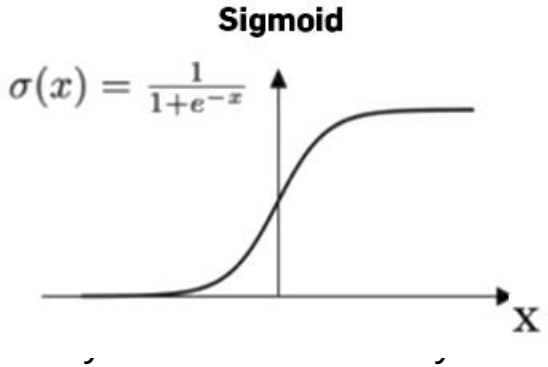
Main activation functions

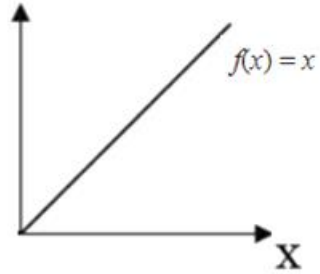
- Sigmoid
  - squishes input into range [0,1]
  - large negative input → 0
  - nice interpretation (neuron active/inactive)
  - PROBLEM: saturates gradients [why?]
  - PROBLEM: output is non-zero centered.

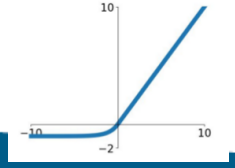
  Mathematically this can creates inefficient updates.
- Tanh
  - Squishes to range [-1,+1]
  - Helps with zig-zagging but still kills the gradients
- ReLu (very popular)
  - Does not saturate if x>0, easy to compute, converges fast
  - Dead ReLus → leaky ReLu max(0.01x,x)
- Exponential Linear Units:
  - similar behaviour as leaky ReLu or rectified ReLu, comput











Source: https://machine-learning.paperspace.com/wiki/activation-function

Cheat sheet for activation functions:

Pros and Cons of Activation Functions

Type of Function	Pros	Cons
Linear	• It gives a range of activations, so it is not binary activation. • It can definitely connect a few neurons together and if more than 1 fire, take the max and decide based on that.	• It is a constant gradient and the descent is going to be on a constant gradient. • If there is an error in prediction, the changes made by backpropagation are constant and not depending on the change in input.
Sigmoid	• It is nonlinear in nature. Combinations of this function are also nonlinear. • It will give an analog activation, unlike the step function.	• Sigmoids saturate and kill gradients. It gives rise to a problem of "vanishing gradients" • The network refuses to learn further or is drastically slow.
Tanh	• The gradient is stronger for tanh than sigmoid i.e. derivatives are steeper.	• Tanh also has a vanishing gradient problem.
ReLU	• It avoids and rectifies the vanishing gradient problem. • ReLu is less computationally expensive than tanh and sigmoid because it involves simpler mathematical operations.	• It should only be used within hidden layers of a Neural Network Model. • Some gradients can be fragile during training and can die. It can cause a weight update which will make it never activate on any data point again. Thus, ReLu could even result in Dead Neurons.
Leaky ReLU	• Leaky ReLUs is one attempt to fix the "dying ReLU" problem by having a small negative slope	• As it possesses linearity, it can't be used for complex Classification. It lags behind the Sigmoid and Tanh for some of the use cases.
ELU	• Unlike ReLU, ELU can produce negative outputs.	• For x>0, it can blow up the activation with the output range of [0, ∞].

Function Type	Equation	Derivative
Linear	$f(x) = ax + c$	$f'(x) = a$
Sigmoid	$f(x) = \frac{1}{1+e^{-x}}$	$f'(x) = f(x)\,(1-f(x))$
TanH	$f(x) = tanh(x) = \frac{2}{1+e^{-2x}} - 1$	$f'(x) = 1 - f(x)^2$
ReLU	$f(x) = \begin{cases} 0 \text{ for } x<0 \\ x \text{ for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 \text{ for } x<0 \\ 1 \text{ for } x \geq 0 \end{cases}$
Parametric ReLU	$f(x) = \begin{cases} \alpha x \text{ for } x<0 \\ x \text{ for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} \alpha \text{ for } x<0 \\ 1 \text{ for } x \geq 0 \end{cases}$
ELU	$f(x) = \begin{cases} \alpha(e^x - 1) \text{ for } x<0 \\ x \text{ for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} f(x) + \alpha \text{ for } x<0 \\ 1 \text{ for } x \geq 0 \end{cases}$

- Source: https://www.analyticsvidhya.com/blog/2021/04/activation-functions-and-their-derivatives-a-quick-complete-guide/

Further topics: Problems with (stochastic) gradient descent

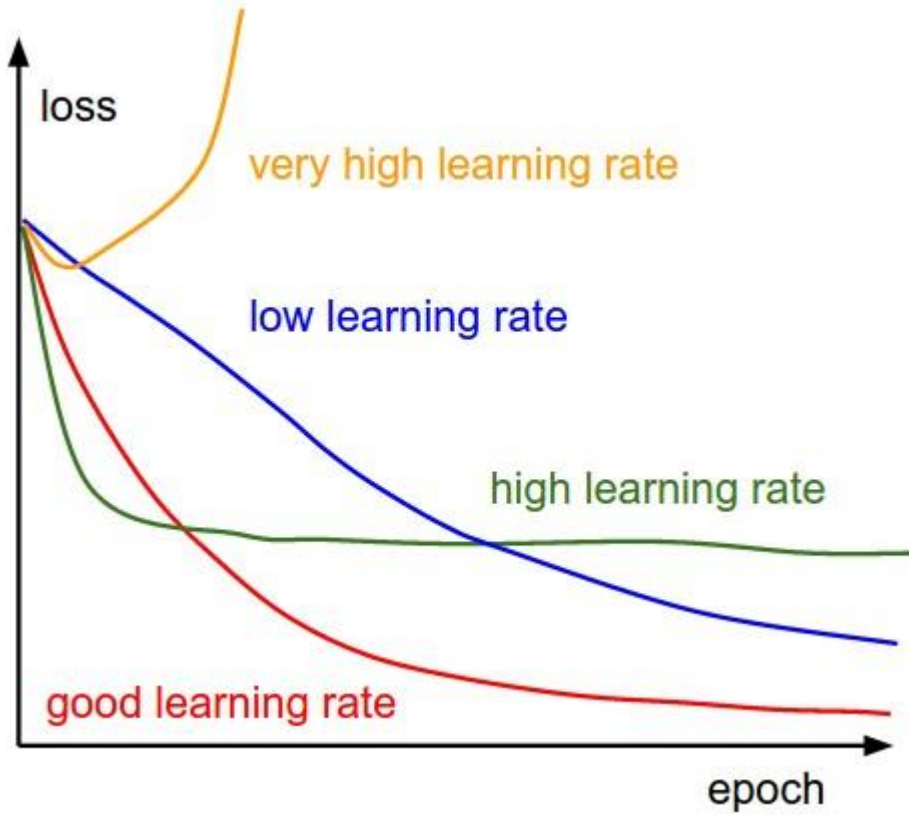
- Cost function changes quickly w.r.t. θ_1 and slow w.r.t. θ_2. The network takes very long to train.
- Cost function gets stuck at a saddle point.
- Often addressed with AdaGrad (modification of step 3):

  (3) Perform the following Gradient Descent

  - $D^{2(l)} = D^{2(l)} + \left(D^{(l)}\right)^2$ ← $D^{(l)}$ *is backprop gradient and* $D^{2(l)}$ *accumulator for squares of gradient*
  - $\Theta_{i+1}^{(l)} = \Theta_i^{(l)} - \left(\frac{\alpha}{\sqrt{D_{i+1}^{2\,(l)}}}\right) D_i^{(l)}$ ← *learning rate is large for* θs *with small gradient + some momentum*

Facilitates parameter-specific adaptive learning rates.

Good learning rate



Source: Stanford's class. See next slide.

Further Topics: Regularization

- Prevents overfitting (pushes weights towards zero).
- The only difference is that the cost function now includes the regularization term:

$$\mathrm{J}(\Theta) = \frac{1}{2m}\sum_{i=1}^{m}\left(a^{(L)} - y_i\right)^2 + \lambda\sum_{j=1}^{n}\theta_j^2$$

- Where λ is the ***regularization parameter*** and n number of parameters excluding bias terms.
- We do not regularize the bias ($j = 1$)!
- Regularizing bias can introduce significant under fitting (=regression through the origin). Bias is very powerful in improving fit and requires little data to estimate.

Regularization – what changes in the estimation?

- Cost function (see previous slide)
- It changes all partial derivatives of the cost function w.r.t. parameters (used in backpropagation!)
- However, regularization term enters in aditive fashion so it is very easy to handle.
- We only need to re-compute the gradient (when back propagation ends) with (only for the weights, not for the bias!):

$$\mathrm{D}_{\mathrm{ij}}^{(l)} = \mathrm{D}_{ij}^{(l)} + \frac{\lambda}{m}\theta_{ij}^{(l)}$$

Dropout regularization

- Alternative way of regularizing the network.
- Needed more for large and deep networks.
- During training some neurons are randomly dropped or “shut down.”
- This prevents the neurons from “adapting” to what other neurons are doing (“units may change in a way that they fix up the mistakes of the other units”, Srivastava et al. (2014)).
- Neurons become more general/robust/independent.

https://jmlr.org/papers/v15/srivastava14a.html

Recommendation: Great overview of these topics and much more in Stanford's lecture series on convolutional NNs

Lecture 6
- activation functions
- initialization of weights
- data normalization

Lecture 7
- Optimization algorithms

Lecture 10
- NN layers

The whole series of lectures is full of practical advice:
https://www.youtube.com/playlist?list=PL3FW7Lu3i5JvHM8ljYj-zLfQRF3EO8sYv

Further Topics: Types of networks often used in forecasting time series

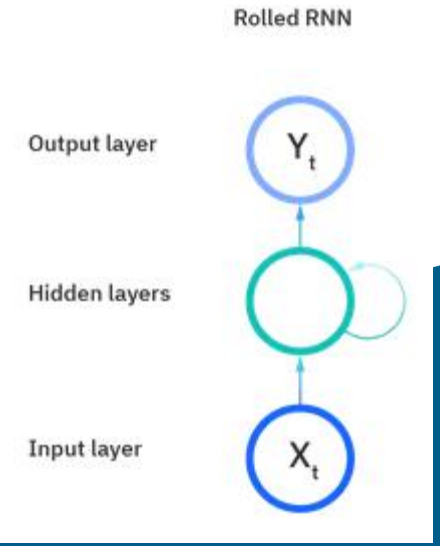
- Recurrent neural network

At every time step a “recurrence formula” is applied. Recurrence formula takes a hidden state from the previous time step and inputs it into a new state (vanilla RNN):

before: $a_t^{(2)} = h(\Theta a_t^{(1)})$

RNN: $a_t^{(2)} = h(\begin{bmatrix} \Theta a_t^{(1)} \\ \gamma a_{t-1}^{(2)} \end{bmatrix})$

They memorize previous inputs which in time series matters because time series patterns are persistent over time. Prone to gradient vanishing or exploding.



Source: www.ibm.com

- Long short-term memory neural network

  - RNN had one hidden state which moved through time.
  - Vanilla RNN has two problems: 1) they have “short-term” memory (the effect of past state on the current and future hidden state vanishes too quickly) and 2) their gradients can be vanishing or exploding.
  - A more sophisticated type of RRN, called LSTM, addresses these two problems.
  - LSTM has two hidden states, the normal «hidden state» (similar to before, $a_t^{(2)}$) and the so-called state called the «cell state» (c_t) which is “hidden” from other parts of the network. “Cell state” acts like a memory cell.
  - The so-called “gates” which control how much of the previous information is kept in the “cell state” (forget gate), how much of the new information should be added to the “cell state” (input gate) and finally what information to output into the new hidden state ($a_t^{(2)}$) (“output gate”).
  - In essence, “cell state” is similar to memory placeholder and the input and forget gate control for how long that memory should be kept or replaced with new information. This architecture allows the NN to model longer dependences.

Further Topics: Some results on (macro) time series vs NN…

«Macroeconomic Indicator Forecasting with Deep Neural Networks», Cook&Hall (2017)

- They forecast US civilian unemployment with 4 NN architectures:
  - Fully connected multi-layer network
  - Convolutional neural network
  - LSTM
  - Encoder-decoder network (upgrade of LSTM)
- Benchmarks: Survey of Professional Forecasters, direct forecasts in an AR model
- Advice: Always add a linear connection from x's to y.
- Findings: for short horizons (0-6m) all NN outperform benchmarks. For longer horizons Encoder-decoder network outperforms benchmarks. Rather large improvements: 50-30% smaller mean absolute error.
- Caution: MAE instead of MRSE (why?). Limited set of benchmark models used.

“How is machine learning useful for macroeconomic forecasting”, Coulombe et al. (2022)

- They do not estimate neural networks. However, they investigate which aspects of machine learning algorithms are crucial for better ML performance. Thus, their results likely apply to the NN-case as well.
- They focus on aspects of ML algorithms which traditional TS models omit: nonlinearities, regularization, cross-validation, alternative loss-functions used in ML.
- Forecast: 5 representative US macroeconomic series (INDPRO, UNRATE, INF, SPREAD, HOUS) over 456 evaluation periods.
- **Findings**: 1) ML non-linearities drive the success of ML methods (non-lin are present in TS but often lack regularization), 2) cross-validation does not seem to help much

“Nowcasting GDP Growth in a Small Open Economy” Marcellino and Sivec (2021)

Overall, mixed frequency dynamic factor models and neural networks perform well, both in absolute terms and in relative terms with respect to a benchmark AR model (10-20% lower RMSE). The gains are larger during problematic times, such as the financial crisis and the recent Covid period. [NN do better but sometimes make weird mistakes…]

“Linear models, smooth transition autoregressions, and neural networks for forecasting macroeconomic time series: A re-examination”, Terasvirta et al. (2005)

- They compare NN against non-linear TS models.
- “..at long forecast horizons, an NN model obtained using Bayesian regularization produces more accurate forecasts...” At shorter horizons NN do not perform better.

“Macroeconomic Forecasting with Neural Network Reinforced Factor Models”, Umbach (2021)

- “The results suggest significant improvements in the forecasting accuracy of four major US macroeconomic time series.”

“Nowcasting New Zealand GDP using machine learning algorithms”, Richardson et al. (2018)

- “…ML models produce point nowcasts that are superior to the simple AR benchmark. The top-performing models such as the support vector machines, Lasso (Least Absolute Shrinkage and Selection Operator) and neural networks are able to reduce the average nowcast errors by approximately 16-18 per cent relative to the AR benchmark.”

NN why are they not used more for macroeconomic forecasting?

- NN are considered a black box compared to TS models. Traceability and explainability of forecasts sometimes matters more. Especially if the difference in average forecast error is 5-10%.

Single-hidden layer neural network with n_0 inputs and n_1 hidden units (the simples possible case!):

$$y_t = \theta_{10}^{(2)} + \sum_{i=1}^{n_1} \theta_{1i}^{(2)} \; h\left(\theta_{0i}^{(1)} + \sum_{j=1}^{n_0} \theta_{ij}^{(1)} x_{jt}\right) + \epsilon_t$$

AR model with 1 exogenous regressor:

$$y_t = \alpha_0 + \alpha_1 y_{t-1} + \beta x_{t-1} + \epsilon_t$$

Class finished