https://www.asimovinstitute.org/neural-network-zoo/

Multilayer and deeper

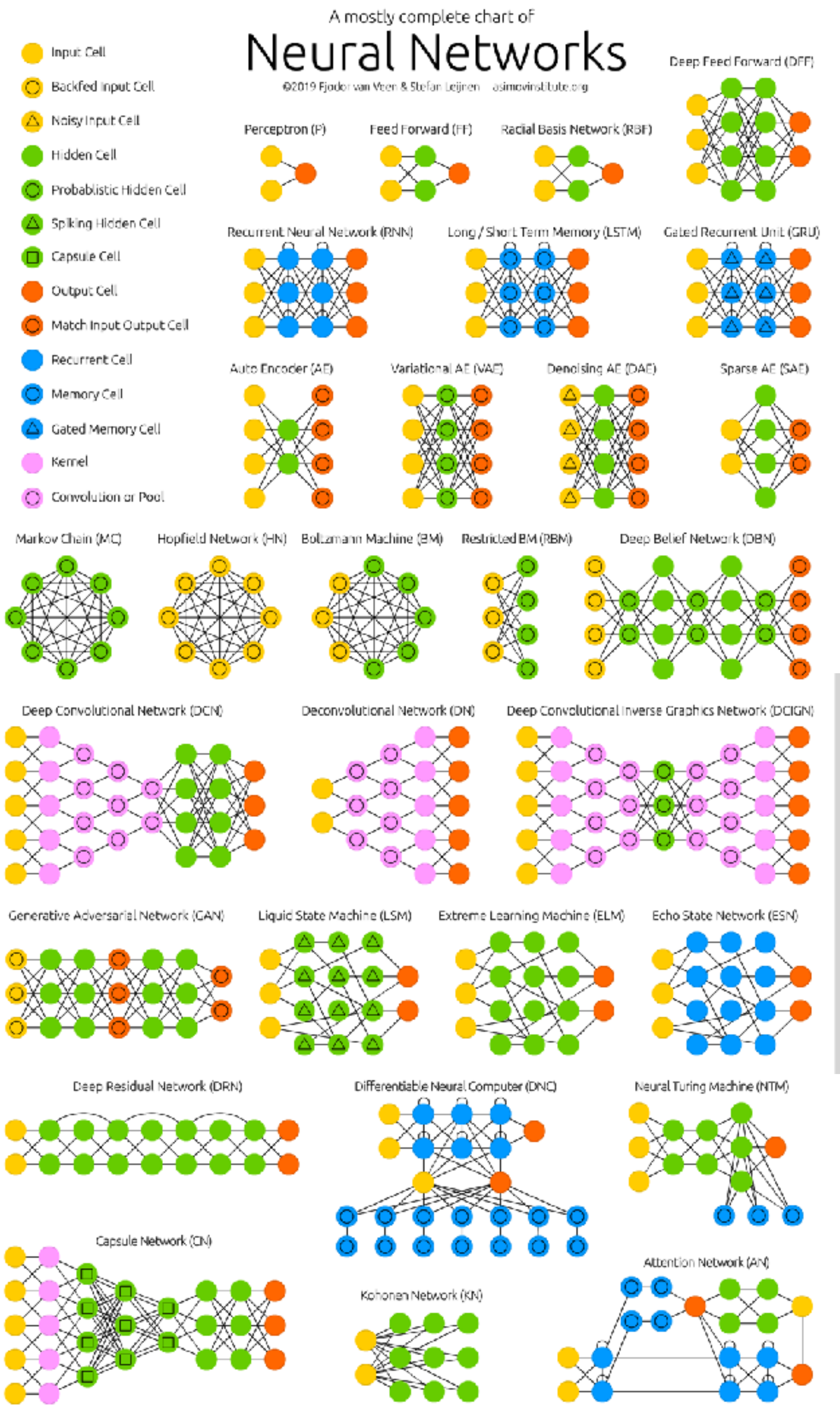
Jason G. Fleischer, Ph.D.

Asst. Teaching Professor
Department of Cognitive Science, UC San Diego

jfleischer@ucsd.edu

@jasongfleischer

https://jgfleischer.com

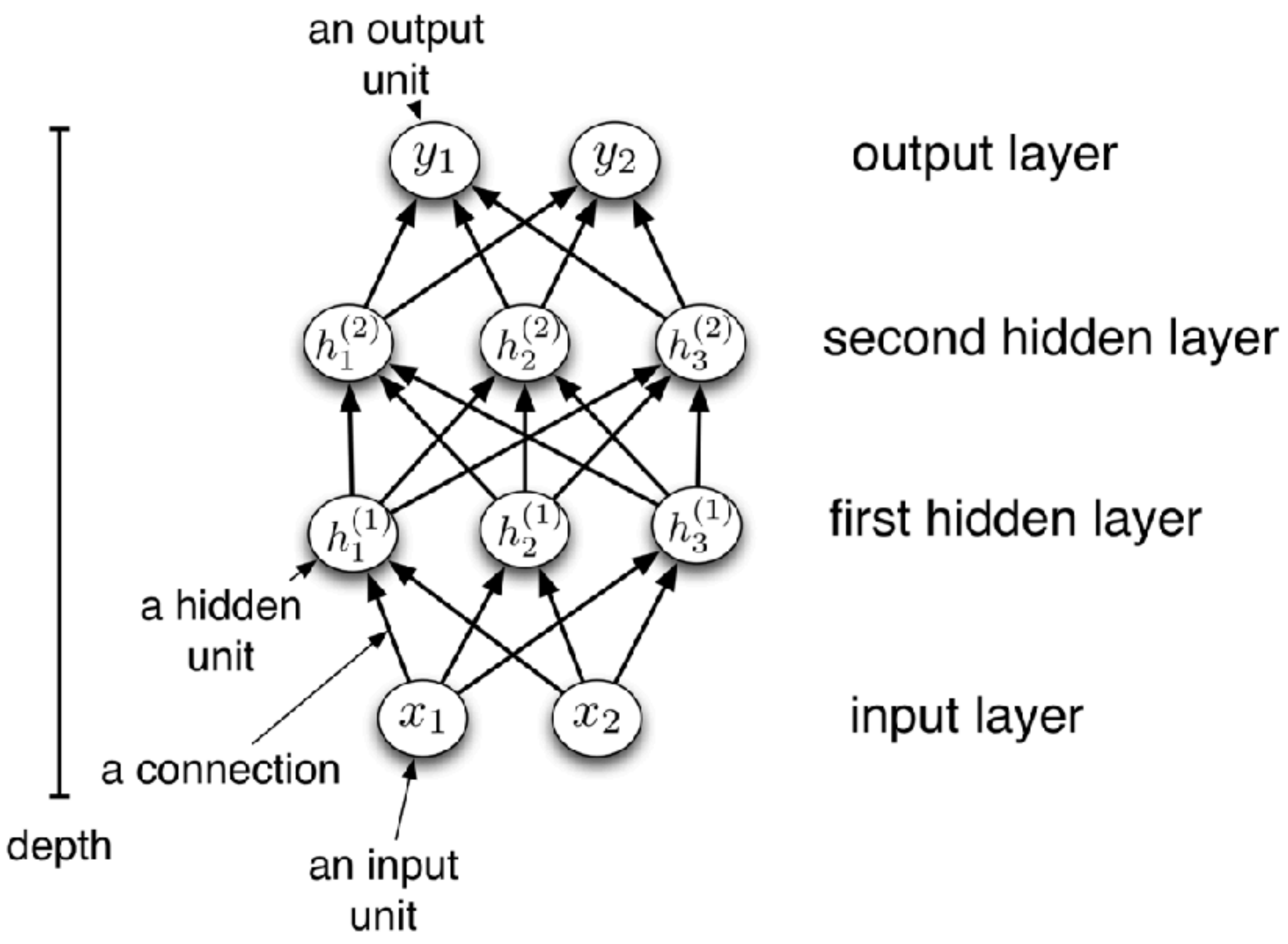


Some slides in this lecture are courtesy of Roger Grosse

Multilayer NNs

A simple example architecture

- D layers deep
- Each layer of M neurons has N inputs from neurons in the previous layer
- Fully connected (all to all)
- MxN weight matrix $\mathbf{W}$
- The M neurons each sum up weighted inputs and run the sum through an activation function (often a non-linearity)



Multilayer NNs

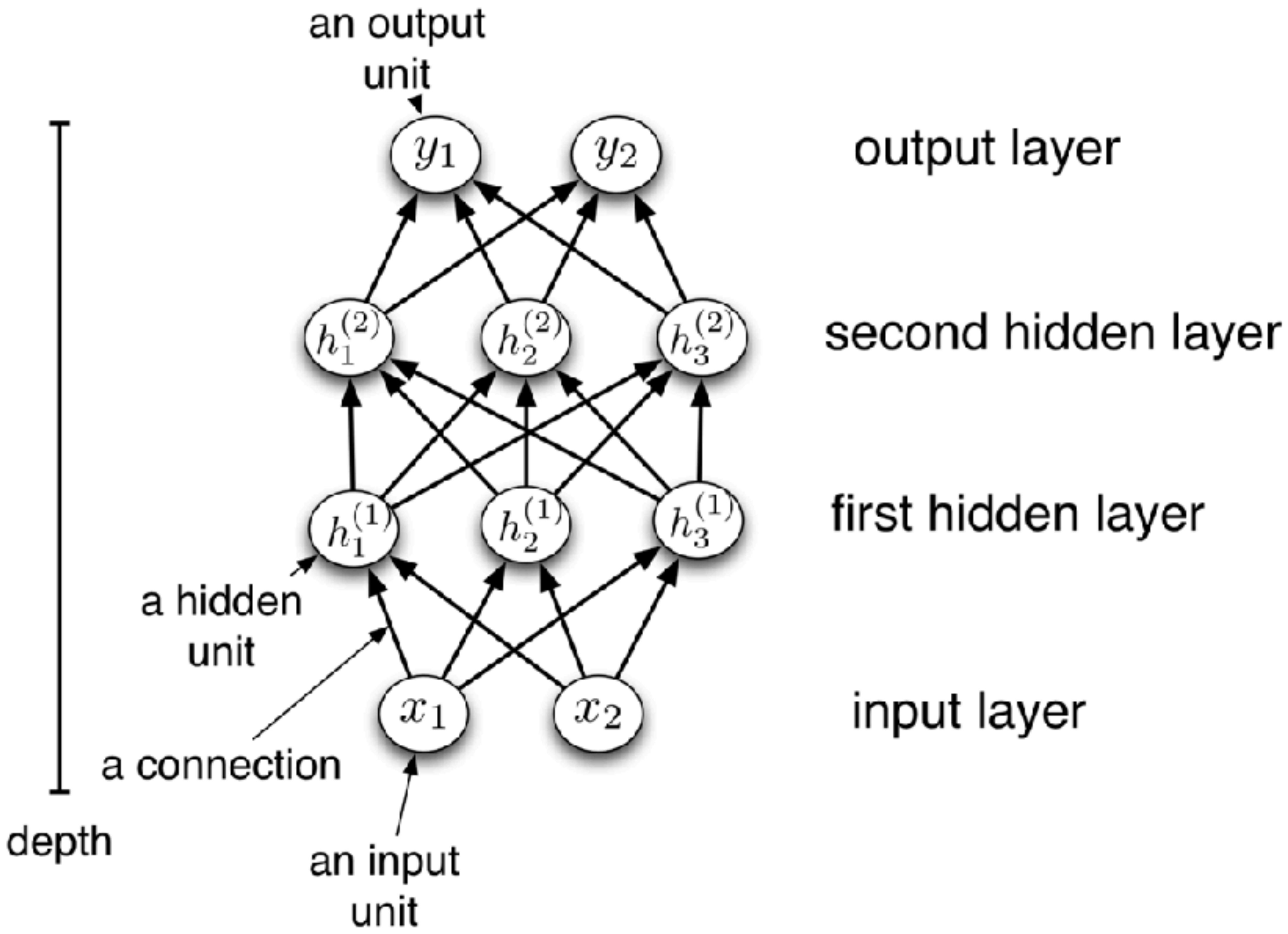
A simple example architecture

- Each layer is a function of the one below it

$$\mathbf{h^{(1)}} = f^{(1)}(\mathbf{x})$$
$$\mathbf{h^{(2)}} = f^{(2)}(\mathbf{h^{(1)}})$$
$$\cdots$$
$$\mathbf{y} = f^{(D)}(\mathbf{h^{(D)}})$$

- Or more simply it is a modular composition of the functions of each layer

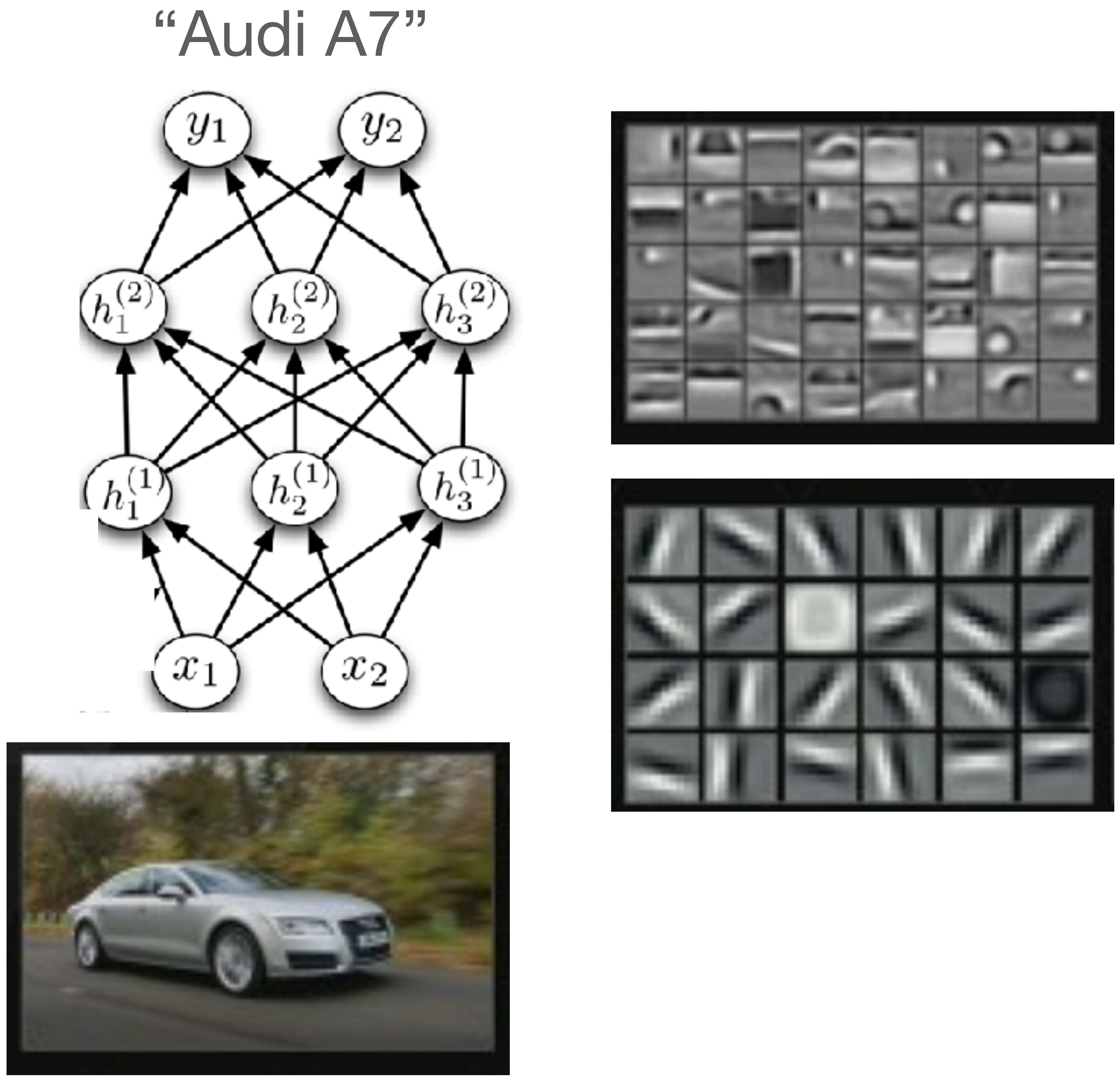
$$\mathbf{y} = f^{(1)} \circ f^{(2)} \ \dots \ \circ f^{(D)}$$



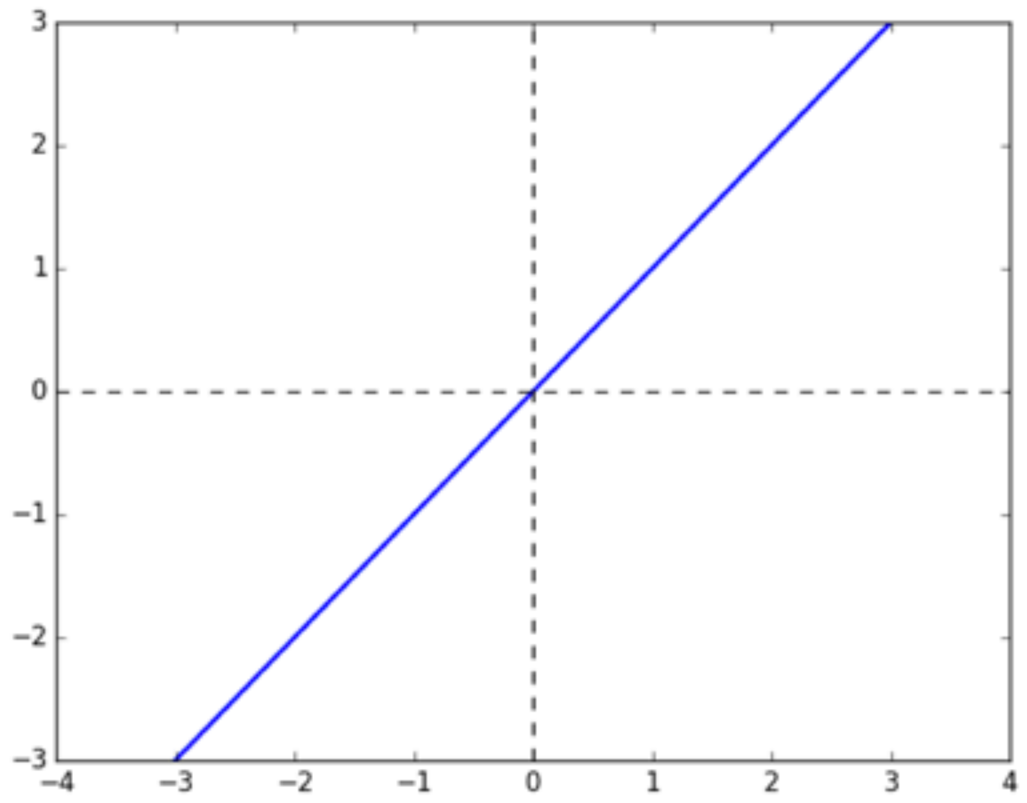
Hidden layers

Self-organized features

- One way to understand what MLPs are doing is to think of each layer as being optimized to provide information to the one above
- In some NN architectures this means that hidden layers are learning feature transformations that enable a better decision at the output

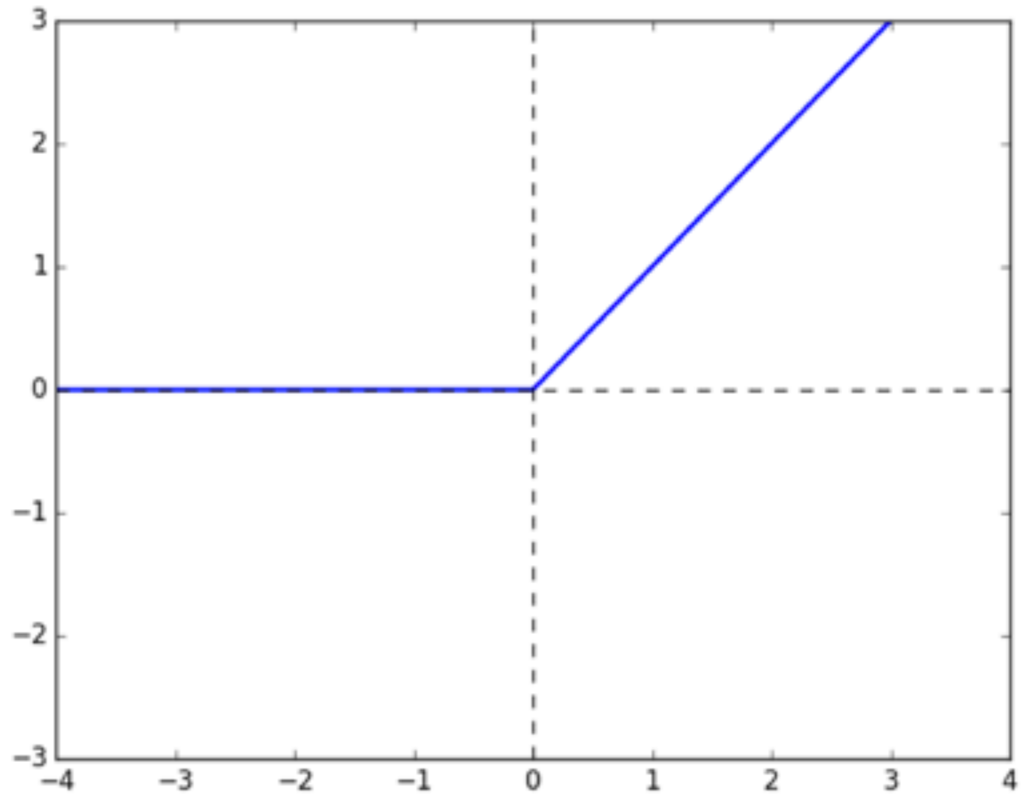


Some activation functions:



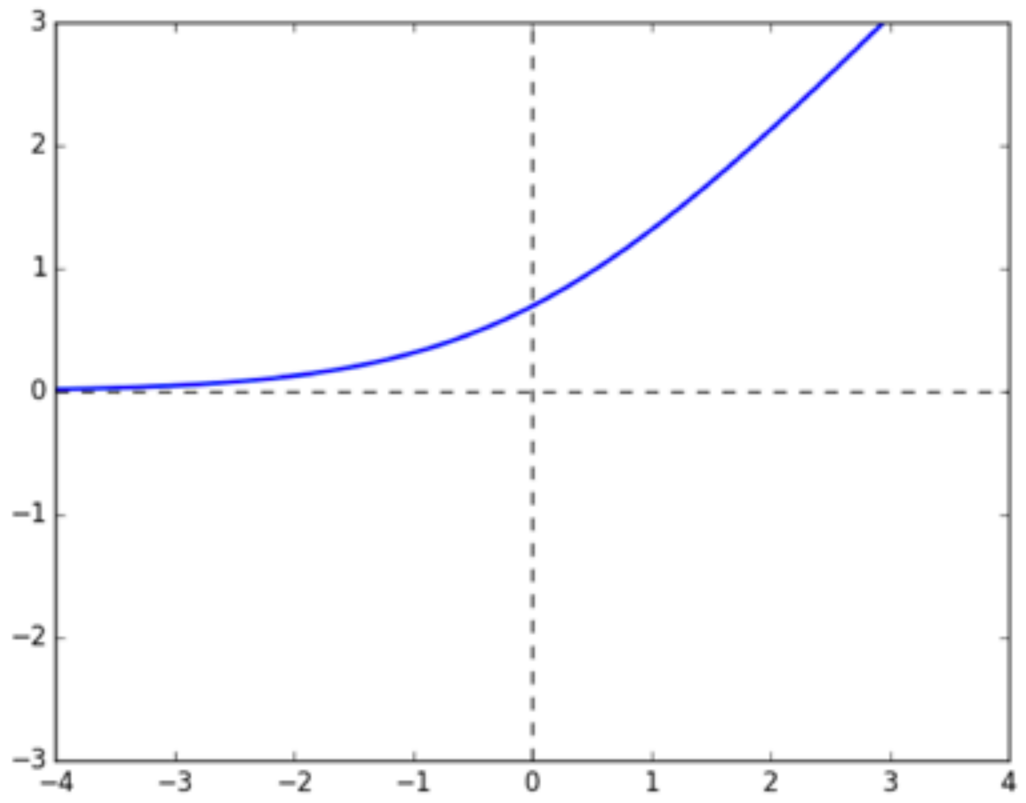
Linear

$$y = z$$



Rectified Linear Unit (ReLU)

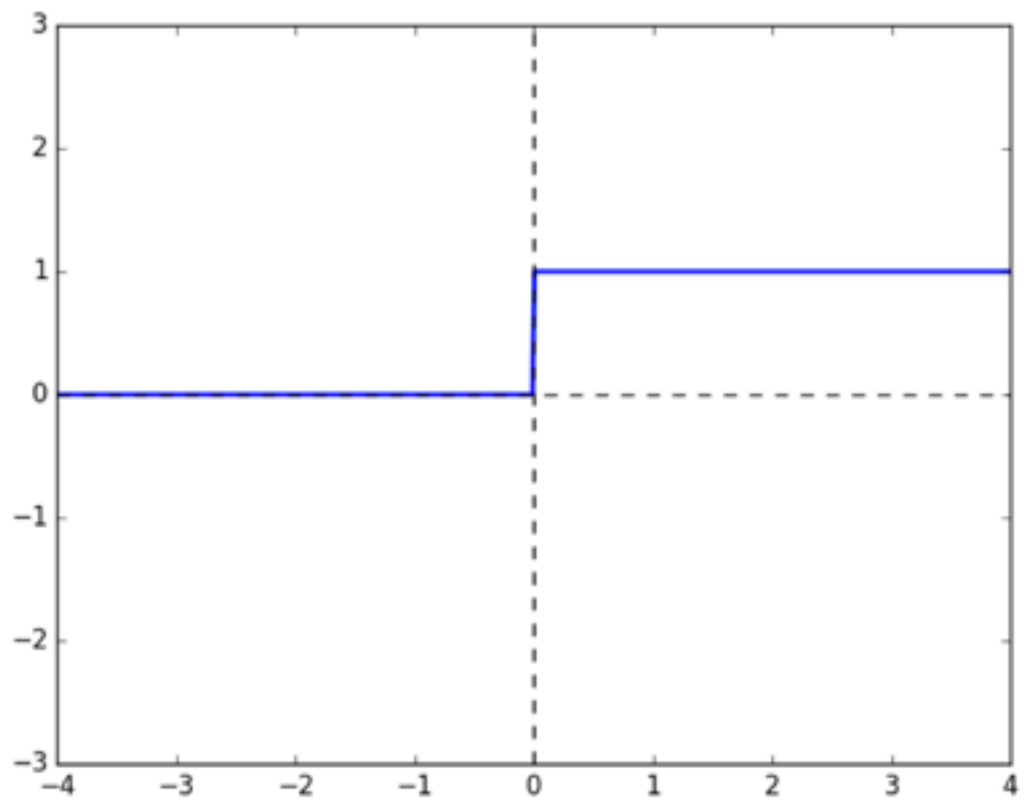
$$y = \max(0, z)$$



Soft ReLU

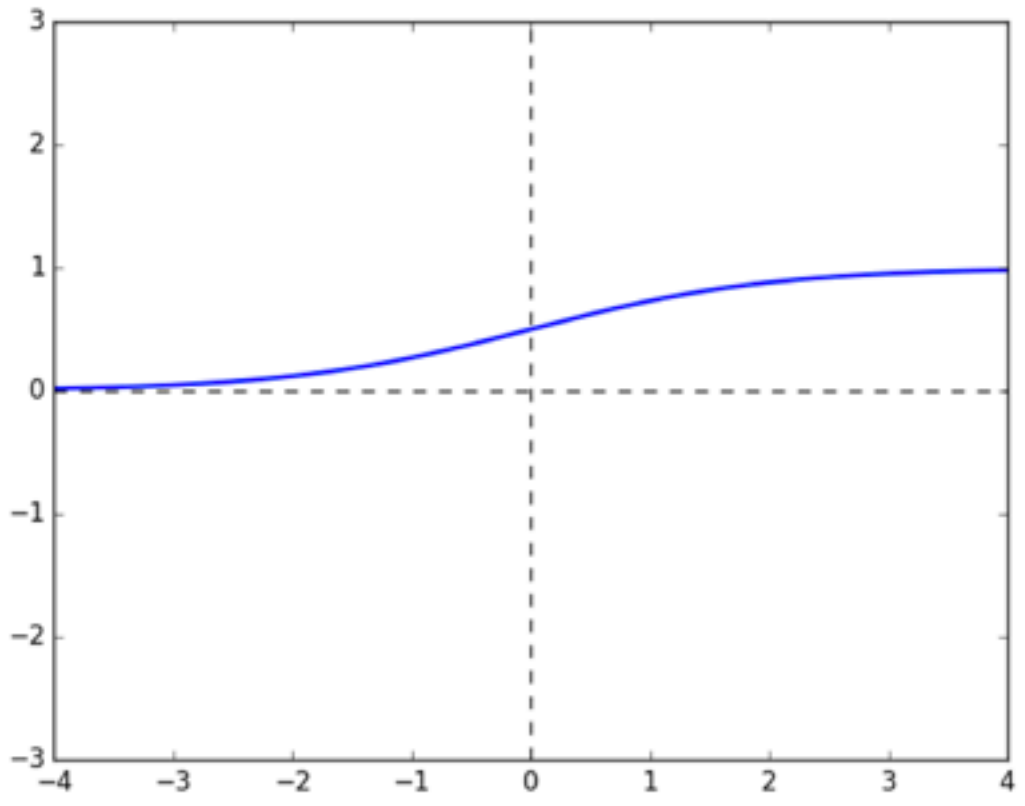
$$y = \log 1 + e^z$$

Some activation functions:



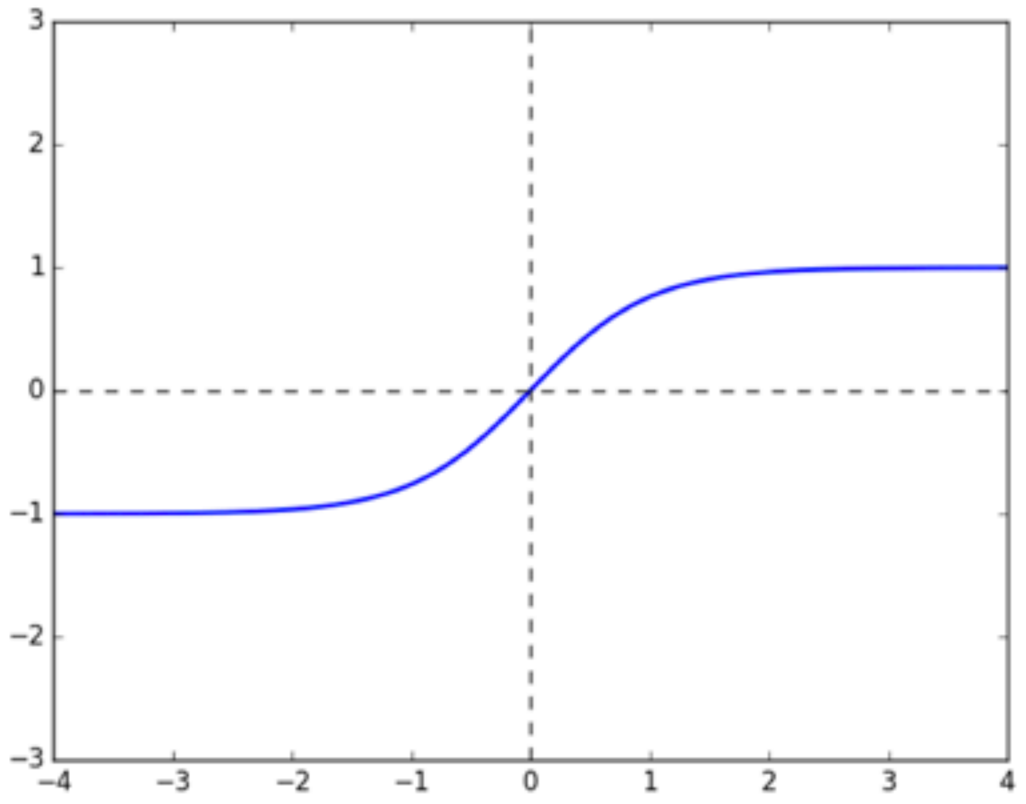
Hard Threshold

$$y = \begin{cases} 1 & \text{if } z > 0 \\ 0 & \text{if } z \leq 0 \end{cases}$$



Logistic

$$y = \frac{1}{1 + e^{-z}}$$



Hyperbolic Tangent (tanh)

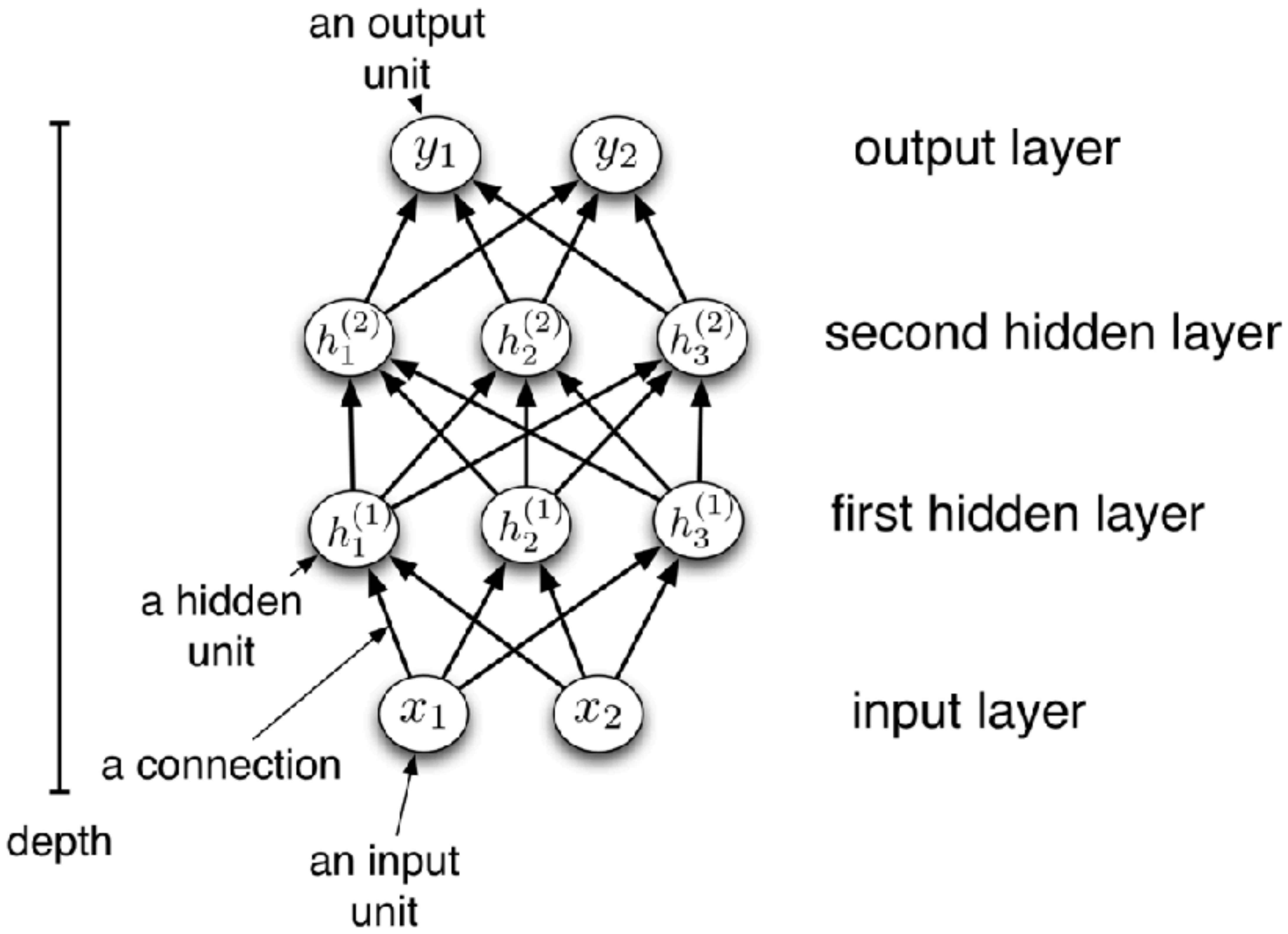
$$y = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$

Picking an activation function

- Hidden layers
  - ReLU for CNN, and more generally ✅
  - Sigmoid, logistic, tanh only for recurrent neural networks, more generally ❌
    - Vanishing gradients can be a problem because the gradients are very similar in the middle
- Output layer based on the type of prediction problem that you are solving:
  - Regression - Linear Activation Function
  - Binary Classification—Sigmoid/Logistic Activation Function
  - Multiclass Classification—Softmax

Multilayer NNs

- MLP are **universal** function approximators if you let them get big enough
  - two layers of arbitrary width with any nonlinearity (even not very nonlinear ones like ReLU)
  - single layers for certain nonlinearities (RBF)



Backprop & automatic differentiation

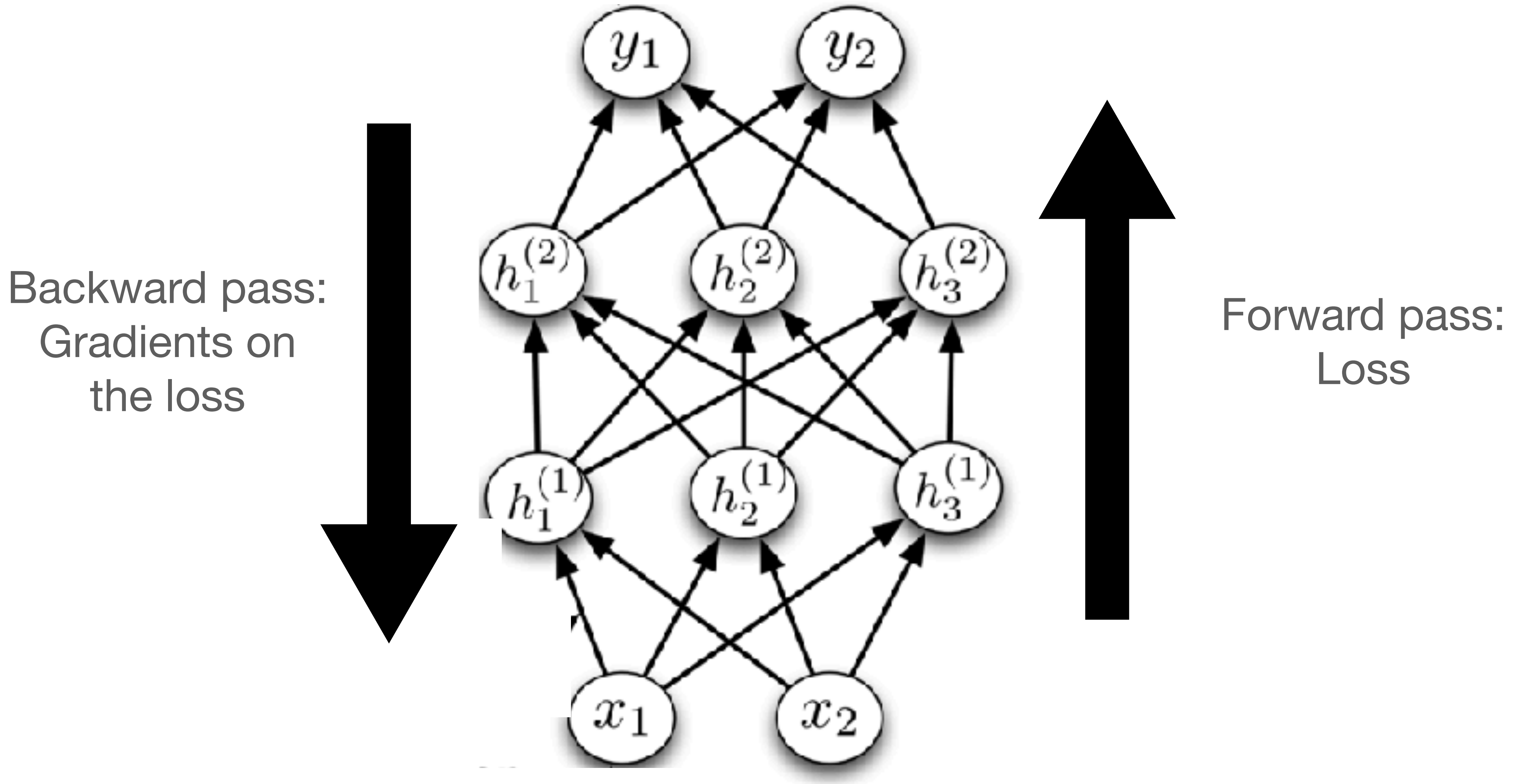
Problem

Assigning responsibility for errors to weights

- For Perceptron (and other algorithms), there are one set of weights. So we can optimize them
- For MLP there’s weights -> nonlinearity -> weights -> nonlinearity -> etc
- Nonlinearity means there’s no lumped equivalent linear system to optimize (can’t do GD on a single super sized weight matrix)
- So…
  - We have errors (loss) at the output layers. We can GD to make those weights better.
  - Need to assign responsibility for the output loss to a previous layer and construct a local loss that we can GD to optimize those weights
  - And keep doing that recursively until we optimize the first set of weights

Backpropogation

- Forward pass: calculate the outputs… and therefore the loss
- Backward pass: calculate the gradients and adjust the weights
    - At each layer using the gradient from the layer above



Univariate Chain Rule $\frac{\partial f}{\partial x} = \frac{\partial f}{\partial r}\frac{\partial r}{\partial x}$

Recall: Univariate logistic least squares model

$$
\begin{aligned}
z &= wx + b \\
y &= \sigma(z) \\
\mathcal{L} &= \frac{1}{2}(y - t)^2
\end{aligned}
$$

Let's compute the loss derivatives.

Univariate Chain Rule $\frac{\partial f}{\partial x} = \frac{\partial f}{\partial r}\frac{\partial r}{\partial x}$

How you would have done it in calculus class

$$
\begin{aligned}
\mathcal{L} &= \frac{1}{2}(\sigma(wx+b)-t)^2 \\
\frac{\partial \mathcal{L}}{\partial w} &= \frac{\partial}{\partial w}\left[\frac{1}{2}(\sigma(wx+b)-t)^2\right] \\
&= \frac{1}{2}\frac{\partial}{\partial w}(\sigma(wx+b)-t)^2 \\
&= (\sigma(wx+b)-t)\frac{\partial}{\partial w}(\sigma(wx+b)-t) \\
&= (\sigma(wx+b)-t)\sigma'(wx+b)\frac{\partial}{\partial w}(wx+b) \\
&= (\sigma(wx+b)-t)\sigma'(wx+b)x
\end{aligned}
$$

$$
\begin{aligned}
\frac{\partial \mathcal{L}}{\partial b} &= \frac{\partial}{\partial b}\left[\frac{1}{2}(\sigma(wx+b)-t)^2\right] \\
&= \frac{1}{2}\frac{\partial}{\partial b}(\sigma(wx+b)-t)^2 \\
&= (\sigma(wx+b)-t)\frac{\partial}{\partial b}(\sigma(wx+b)-t) \\
&= (\sigma(wx+b)-t)\sigma'(wx+b)\frac{\partial}{\partial b}(wx+b) \\
&= (\sigma(wx+b)-t)\sigma'(wx+b)
\end{aligned}
$$

What are the disadvantages of this approach?

Univariate Chain Rule

A more structured way to do it

Computing the loss:

$$z = wx + b$$
$$y = \sigma(z)$$
$$\mathcal{L} = \frac{1}{2}(y - t)^2$$

Computing the derivatives:

$$\frac{\mathrm{d}\mathcal{L}}{\mathrm{d}y} = y - t$$
$$\frac{\mathrm{d}\mathcal{L}}{\mathrm{d}z} = \frac{\mathrm{d}\mathcal{L}}{\mathrm{d}y}\,\sigma'(z)$$
$$\frac{\partial \mathcal{L}}{\partial w} = \frac{\mathrm{d}\mathcal{L}}{\mathrm{d}z}\,x$$
$$\frac{\partial \mathcal{L}}{\partial b} = \frac{\mathrm{d}\mathcal{L}}{\mathrm{d}z}$$

Remember, the goal isn't to obtain closed-form solutions, but to be able to write a program that efficiently computes the derivatives.

Univariate Chain Rule

- We can diagram out the computations using a computation graph.
- The nodes represent all the inputs and computed quantities, and the edges represent which nodes are computed directly as a function of which other nodes.