1) why Functional Programming (& Haskell)?

2) models/equations for Neural Networks?

3) results/demo?

vs. TensorFlow, PyTorch → External Library (size? Mo? performance?)

$\neq$

Haskell → Native (Ko ?)

1) Why Functional Programming (& Haskell)?

- declarative, higher-order functions (functions as parameters)
  ↳ no affectation, no loop .. (recursion) ⇒ shorter/compact

Eg.

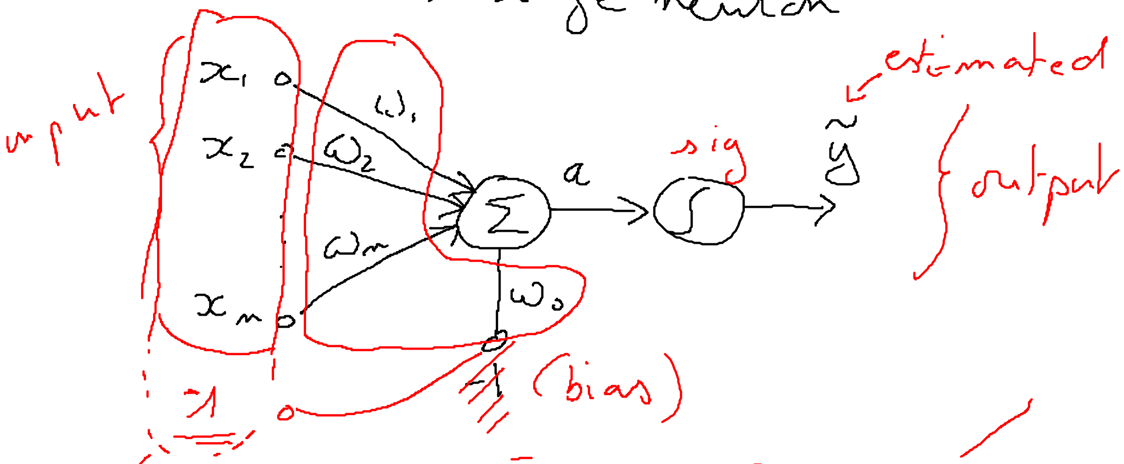
madd =
zipWith add == addition of matrixes !

zipWith add $[x_1, \dots x_n]$ $[y_1 \dots y_n]$
== [add x_1 y_1, ... add x_n y_n]
↳ add de 2 vectors

- ↳ zipWith (*) $[x_1, \dots, x_n]$ $[y_1 \dots y_n]$ == $[x_1 * y_1, \dots x_n * y_n]$
  mult = zipWith (*) -- prod of 2 lists/vectors
  add = zipWith (+)
  minus = zipWith (-)
- map f $[x_1, \dots x_n]$ == $[f\ x_1, \dots, f\ x_n]$
  ↳ sig (x) = 1 / (1 + e(-x))
  ↳ $\boxed{\text{s}}$igmoid = map sig -- sigmoid of a vector !
- foldr1 (+) $[x_1, \dots x_n]$ == $x_1 + x_2 + \dots + x_n$
  ↳ sum = foldr1 (+)

2) models/equations for Neural Networks ?

a) single neuron



$ws = [\omega_0, \omega_1, \dots \omega_m]$

$a = \underbrace{\Sigma(x_0\omega_0 + x_1\omega_1 + \dots x_m \omega_m)}$

$a = X \cdot W$ dot

//recall ws xs =

a = sum (mult ws ys)

$y\emptyset$ = sig (a)

vector $xs = [x_1, \dots x_m]$

shift ~~xs~~, = -1 : ~~xs~~,

learning/adaption ?

- desired output y
- error $e = \frac{1}{2}(y - \tilde{y})^2$
- error $= f(\tilde{y})$, $\tilde{y} = g(a)$, $a = h(w)$

$\frac{\partial e}{\partial \tilde{y}} = \tilde{y} - y$ $\frac{\partial \tilde{y}}{\partial a} = sig'(a)$ with $\boxed{sig'(x) = sig(x) * (1 - sig(x))}$

$(*)$ $\frac{\partial a}{\partial w} = \tilde{X}$ $= xs$ $\Rightarrow \frac{\partial e}{\partial w} = \underbrace{\frac{\partial e}{\partial \tilde{y}} \cdot \frac{\partial \tilde{y}}{\partial a}}_{\text{scalar}} \odot \underbrace{\frac{\partial a}{\partial w}}_{\text{vector}}$

de = y0 - y

dy0 = sig' a

da = shift xs

dw = multv (de * dy0) xs

multv k xs = [k.x1 , ... , k.xn]

= map (k*) xs

learning rate (eg. 0.1)

$$ws' = \underbrace{ws}_{\text{vector}} - \underbrace{k}_{\text{scalar}} \underbrace{\frac{\partial e}{\partial ws}}_{\text{vector (dw)}}$$

$k = 0.1$

In Haskell: `ws' = minus ws (multo k dw)`

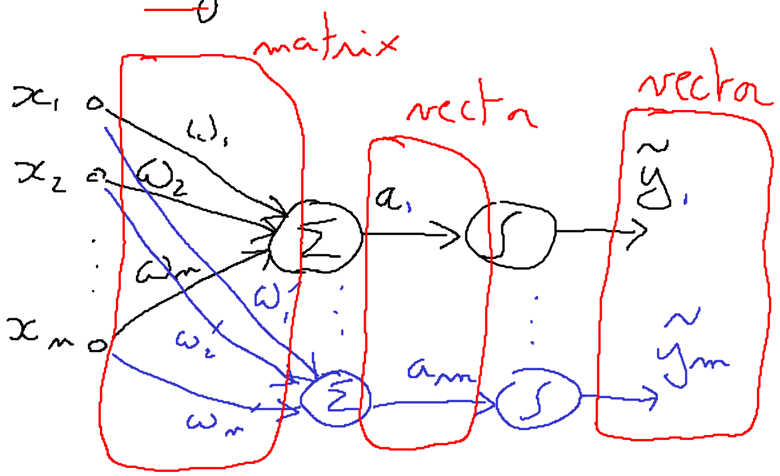
Or `f x y == x `f` y` (syntactic sugar)

`ws' = ws `minus` (k `multo` dw)` similar

Or `(<->) = minus`   `<.> = multo`

`ws' = ws <-> (k <.> dw)`

b) Layer



$$\underbrace{a}_{\text{vector}} = \underbrace{ws}_{\text{mat}} \odot \underbrace{xs}_{\text{vector}}$$

?

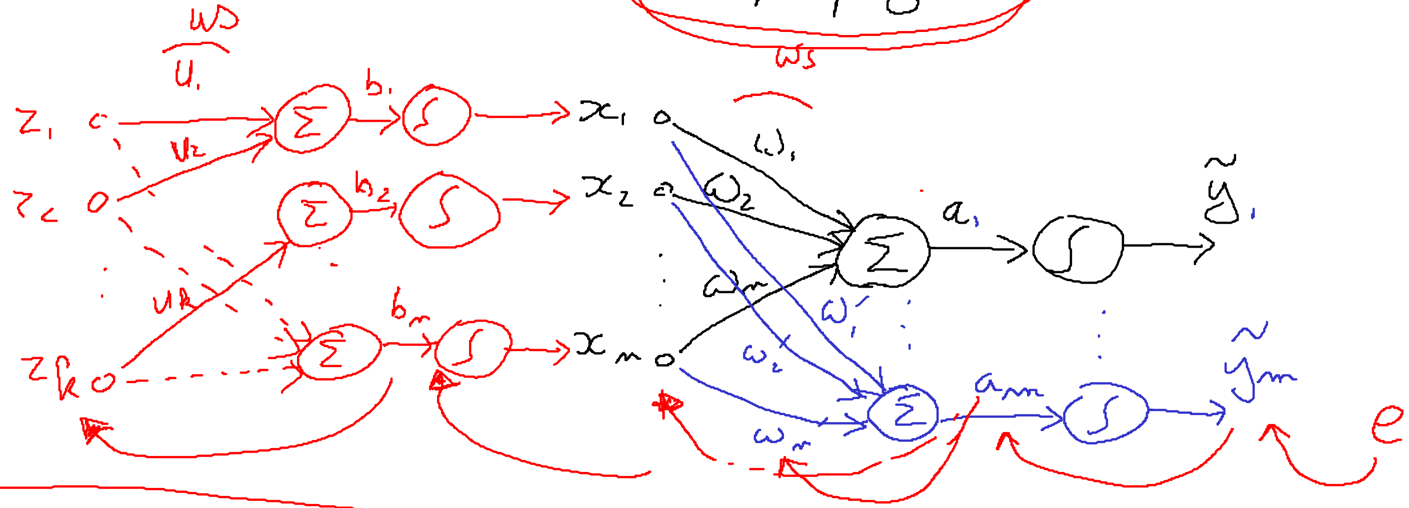
$y\emptyset = \text{sigmoid } a$

$e = \ldots$

$\frac{\partial e}{\partial a} = \ldots$

$dw = \ldots$

c) general model & backpropagation



In Haskell...

dw

(*) $\frac{\partial a}{\partial x} = \omega$

$\frac{\partial x}{\partial b} = sig'(b)$

$\frac{\partial b}{\partial u} = z \Rightarrow \frac{\partial e}{\partial u} = \frac{\partial e}{\partial \tilde{y}} \frac{\partial \tilde{y}}{\partial a} \frac{\partial a}{\partial x} \frac{\partial x}{\partial b} \frac{\partial b}{\partial u} \Rightarrow u' = u - k * \frac{\partial e}{\partial u}$

3) results / demo ?