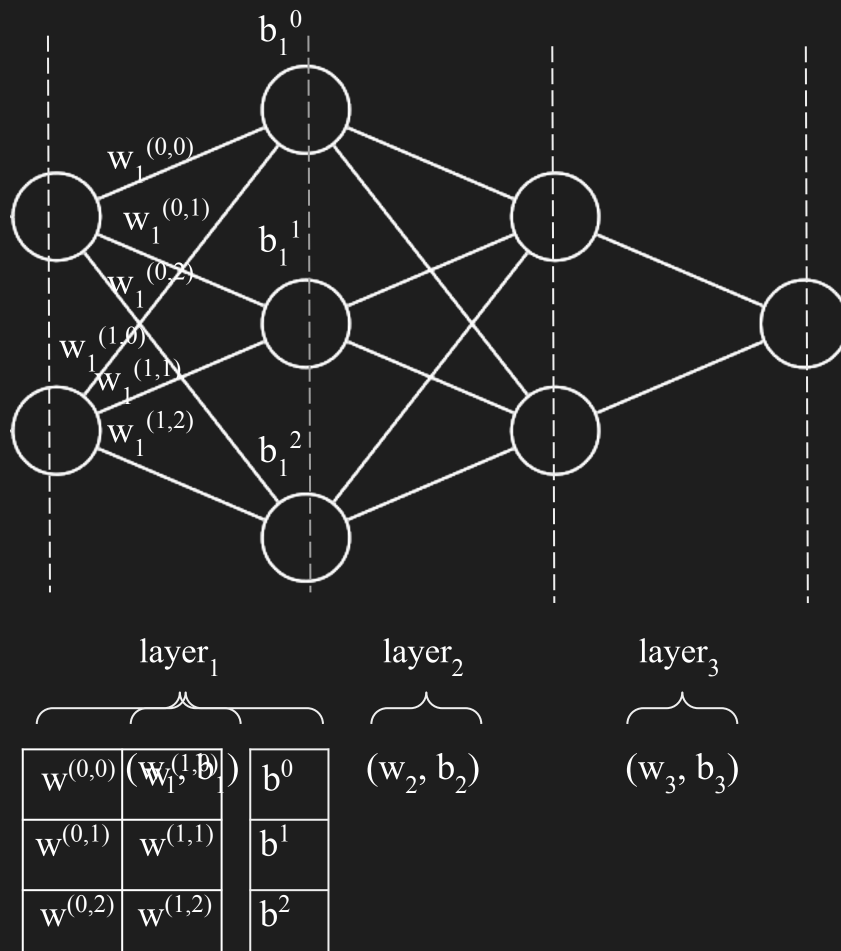
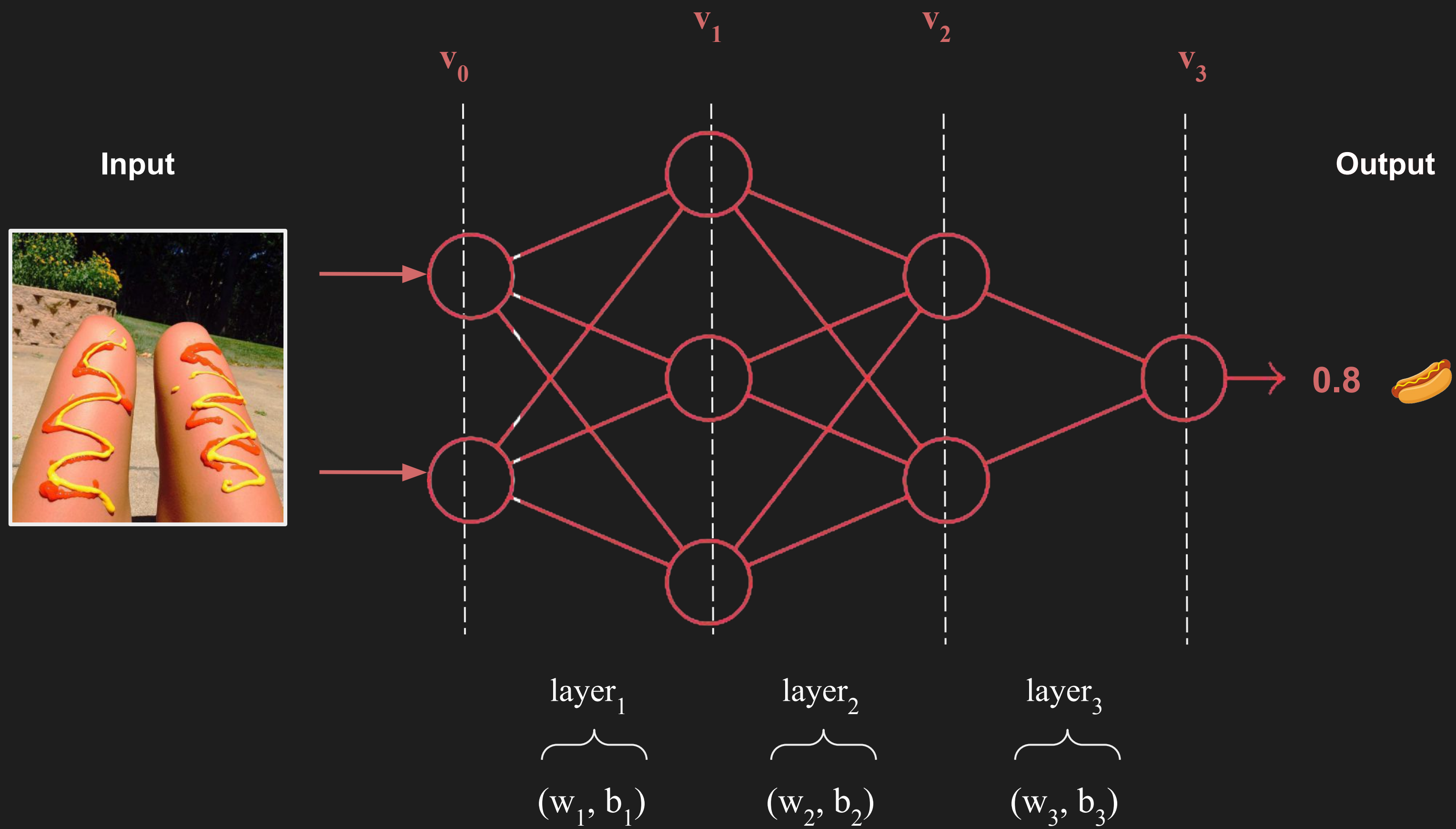
Folding over Neural Networks

Minh Nguyen & Nicolas Wu

Neural networks: structure



Neural networks: forward propagation

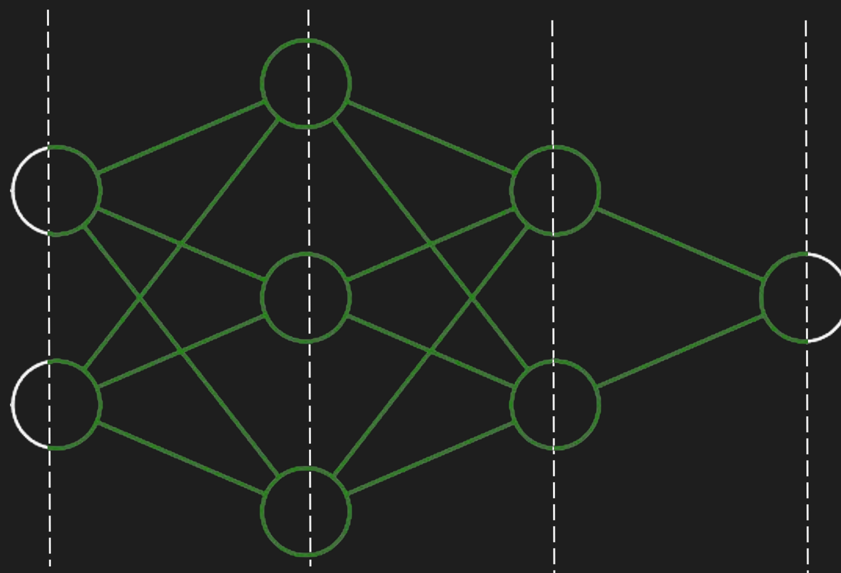


$$\boxed{v_n = \sigma(w_n * v_{n-1} + b_n)}$$

Neural networks: back propagation

$\delta_1 = \sigma^{-1}(\delta_2)$ $\delta_2 = \sigma^{-1}(\delta_3)$ $\delta_3 = \sigma^{-1}$(**Output - Desired Output**)

Input



Output **Desired Output**

0.8 🌭 **0** 🌭

layer_1 (w_1, b_1)

layer_2 (w_2, b_2)

layer_3 (w_3, b_3)

$$\boxed{\delta_n = \sigma^{-1}(\delta_{n+1})}$$

Ways to traverse a neural network

By chaining calls between layers

```
class DenseLayer(...):
 ...
 def call(self, inputs):
   return matmul(inputs, self.weights)
```

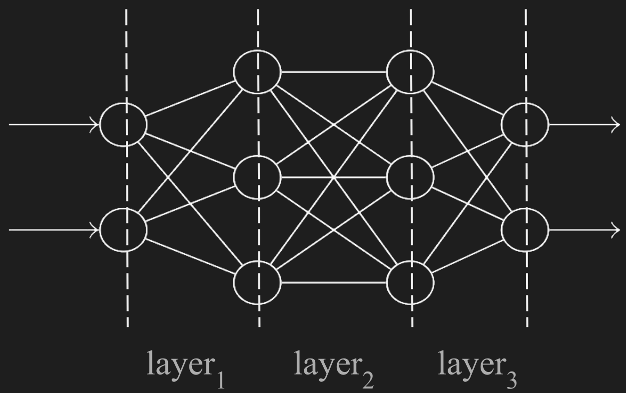
```
layer1 = DenseLayer(..)
layer2 = DenseLayer(..)
layer3 = DenseLayer(..)

output = layer3(layer2(layer1(input)))
```

By explicitly iterating over layers

```
class NeuralNetwork(...):

  def call(...):
    for layer in self.layers:
      ...
```

```
nn = NeuralNetwork()
nn.add(layer1)
nn.add(layer2)
nn.add(layer3)

output = nn(input)
```

By folds?

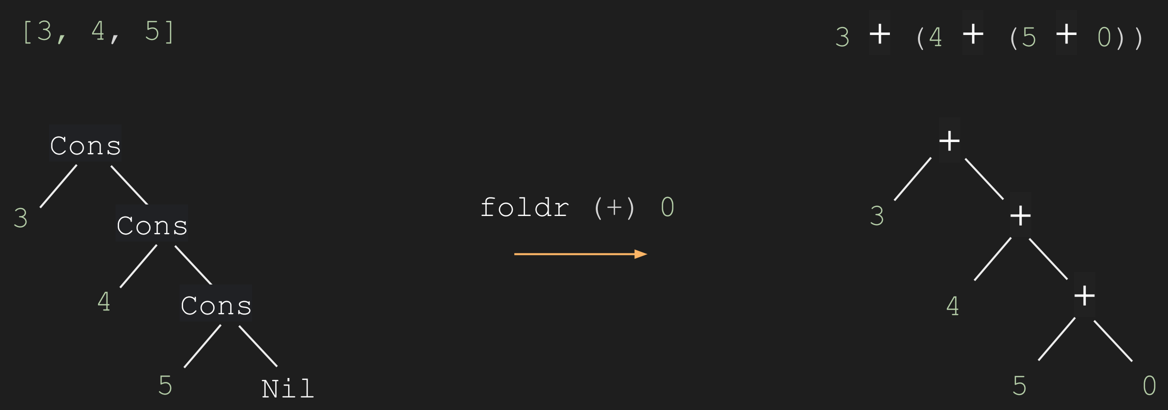



```
type NeuralNet = List Layer

data List a = Cons a (List a)
            | Nil
```

Foldr

As a ***definition***:

```
foldr :: (a → b → b) → b → List a → b
foldr f b0 Nil            = b0
foldr f b0 (Cons x xs) = f x (foldr f b0 xs)
```



As an ***abstraction***, `foldr` lets us decouple recursion from computation

Summing via standard recursion

```
sum :: List Int → Int
sum Nil        = 0
sum (Cons x xs) = x + sum xs
```

Summing via folds

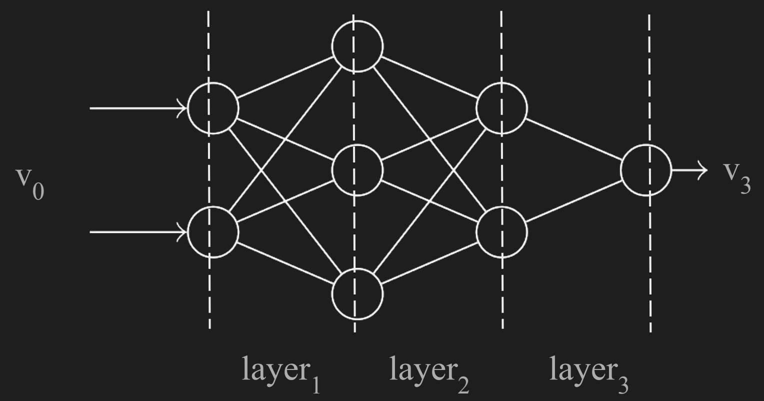
```
sum :: List Int → Int
sum = foldr (+) 0
```

recursion   computation

Foldr for forward propagation

What does forward propagation with **`foldr`** look like?

```
foldr :: (Layer → Values → Values)
      → Values
      → List Layer
      → Values
```



The **elements** are ***layers***

```
type Layer  = (Weights, Biases)
```

```
type Biases   = List Double
type Weights  = List (List Double)
```

The **accumulators** are the ***inputs/outputs*** of layers

```
type Values = List Double
```

The **binary operation** is ***forward propagation*** over a single layer

```
fwd :: Layer → Values → Values
```

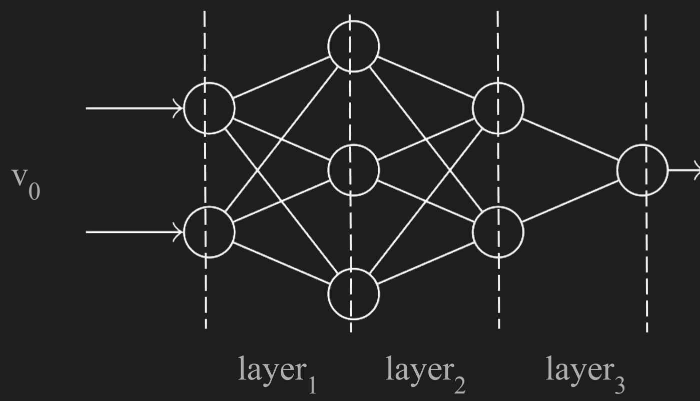
fwd (w_n, b_n) v_{n-1} = $\sigma(w_n * v_{n-1} + b_n)$

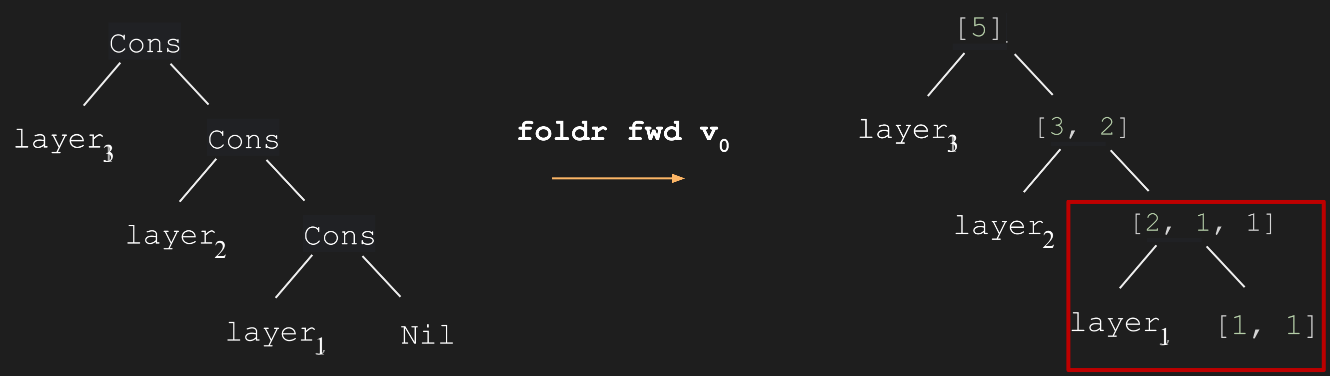
Folding with `fwd` evaluates a neural network to a *forward propagation function*

```
foldr fwd :: List Layer → (Values → Values)
```

Foldr for forward propagation: Example

```
let layer₁ = ( [[1, 1], [0, 1], [0, 1]] , [0, 0, 0])
    layer₂ = ( [[2, 1, 1], [1, 1, 1]]   , [0, 0])
    layer₃ = ( [[1, 1]]                 , [0])
    nn     = [layer₃, layer₂, layer₁]
    v₀     = [1, 1]
in  foldr fwd v₀ nn
```





Reviewing foldr (1)

Suggestion 1
We'd like to avoid needing lists as an intermediate data type

```
foldr :: (Layer → Values → Values) → Values → List Layer → Values
```

Solution
Let's define neural networks in terms of `Layer`

```
data List a   = Cons a (List a)
              | Nil

type Layer  = (Weights, Biases)
```

```
data Layer    = DenseLayer Weights Biases Layer
              | InputLayer
```

Reviewing foldr (2)

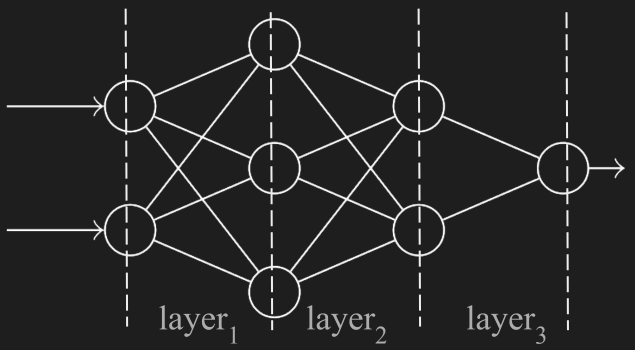
Suggestion 2
We'd like to keep recursion out of the data type for layer:

```
data Layer   = DenseLayer Weights Biases Layer
             | InputLayer
```

Solution
We abstract away the recursive occurrence into a type parameter, `k`

```
data Layer k = DenseLayer Weights Biases k
             | InputLayer deriving Functor
```

```
DenseLayer w3 b3
  (DenseLayer w2 b2
    (DenseLayer w1 b1
      InputLayer))  :: Layer (Layer (Layer (Layer k)))
```



… and then relocate recursion into a different data type, `Fix`

```
data Fix f  = In (f (Fix f))
```

```
In (DenseLayer w3 b3
  (In (DenseLayer w2 b2
    (In (DenseLayer w1 b1
      (In InputLayer))))))  :: Fix Layer
```

Cata for forward propagation

How do we fold a structure of `Fix f`?

```
In (DenseLayer w3 b3
 (In (DenseLayer w2 b2
   (In (DenseLayer w1 b1
     (In InputLayer))))))  :: Fix Layer
```

```
data Layer k = DenseLayer Weights Biases k
             | InputLayer
```

If `foldr` generalises *recursive evaluation* over `List a` …

```
foldr :: (a → b → b) → b → List a → b
foldr fwd v0 Nil          = v0
foldr fwd v0 (Cons x xs) = fwd x (foldr fwd v0 xs)
```

Cata for forward propagation

How do we fold a structure of `Fix f`?

```
In (DenseLayer w3 b3
 (In (DenseLayer w2 b2
   (In (DenseLayer w1 b1
     (In InputLayer)))))) :: Fix Layer
```

```
data Layer k = DenseLayer Weights Biases k
             | InputLayer
```

If `foldr` generalises *recursive evaluation* over `List a` , then `cata` generalises *recursive evaluation* over `Fix f`

```
foldr :: (a → b → b) → b → List a → b
foldr fwd v0 Nil          = v0
foldr fwd v0 (Cons x xs)  = fwd x (foldr fwd v0 xs)
```

```
cata :: Functor f => (f b → b) → Fix f → b
cata fwd = fwd ∘ fmap (cata fwd) ∘ out
```

```
out (In f) = f
```

cata fwd

Cata for forward propagation

How do we fold a structure of `Fix f`?

```
data Layer k = DenseLayer Weights Biases k
             | InputLayer
```

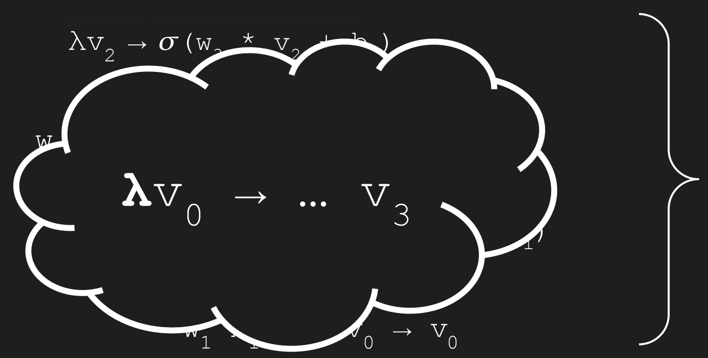
```
In (DenseLayer w_3 b_3
 (In (DenseLayer w_2 b_2
   (In (DenseLayer w_1 b_1
     (In InputLayer)))))) :: Fix Layer
```

If `foldr` generalises *recursive evaluation* over `List a`, then `cata` generalises *recursive evaluation* over `Fix f`

```
cata :: Functor f => (f b → b) → Fix f → b
cata fwd = fwd ∘ fmap (cata fwd) ∘ out
```

We just need to redefine `fwd`:

```
fwd :: Layer (Values → Values) → (Values → Values)
fwd InputLayer = id
fwd (DenseLayer w_n b_n fwd_{n-1}) = (λv_{n-1} → σ(w_n * v_{n-1} + b_n)) ∘ fwd_{n-1}
```



cata fwd

Cata for forward propagation

```
cata fwd :: Fix Layer → (Values → Values)
```

The abstractions we're provided:

```
data Fix f  = In (f (Fix f))
```

Encodes recursive structures

```
cata :: (f b → b) → Fix f → b
cata fwd = fwd ∘ fmap (cata fwd) ∘ out
```

Encodes recursive evaluation

What we had to define:

```
data Layer k = DenseLayer Weights Biases k
             | InputLayer
```

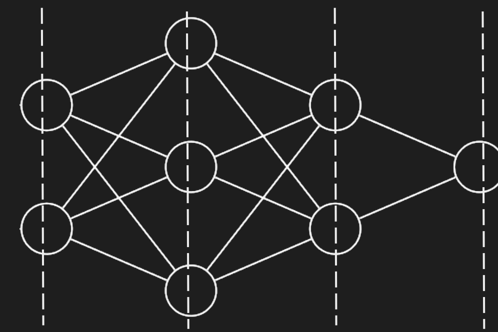
No recursion!

```
fwd :: Layer (Values → Values)
    →        (Values → Values)
fwd InputLayer                = id
fwd (DenseLayer w_n b_n fwd_n-1) = (λv_n-1 → σ(w_n * v_n-1 + b_n)) ∘ fwd_n-1
```

No recursion!

Cata for back propagation

$$\boxed{v_n = \sigma(w_n * v_{n-1} + b_n)}$$



We have **forward propagation**:

```
fwd :: Layer (Values → Values) → (Values → Values)
fwd InputLayer                    = id
fwd (DenseLayer wₙ bₙ fwdₙ₋₁) = (λvₙ₋₁ → σ(wₙ * vₙ₋₁ + bₙ)) ∘ fwdₙ₋₁
```

Can we do the same for **back propagation**?
It takes the error δ_{n+1} of the $(n+1)^{th}$ layer…to update the previous *n* layers.

$$\boxed{\boldsymbol{\delta_n = \sigma^{-1}(\delta_{n+1})}}$$

```
bwd :: Layer (Error → Fix Layer) → (Error → Fix Layer)
bwd (DenseLayer wₙ bₙ bwdₙ₋₁) = λδₙ₊₁ →
      let δₙ    = σ⁻¹(δₙ₊₁)     } compute error
          wₙⁿᵉʷ = wₙ ⊗ δₙ     }
          bₙⁿᵉʷ = bₙ ⊖ δₙ     } update parameters
      in  In (DenseLayer wₙⁿᵉʷ bₙⁿᵉʷ (bwdₙ₋₁ δₙ))
```

$\underbrace{(bwd_{n-1}\ \delta_n)}_{\textit{update the previous n-1 layers}}$

```
bwd InputLayer = λ_ → In InputLayer
```

```
type Error  = [Double]
```

Training neural networks as a catamorphism

We have **forward propagation**:

```
fwd :: Layer (Values → Values)
     →       (Values → Values)

cata fwd :: Fix Layer → (Values → Values)
```

and **back propagation**:

```
bwd :: Layer (Error → Fix Layer)
     →       (Error → Fix Layer)

cata bwd :: Fix Layer → (Error → Fix Layer)
```

"Any pair of folds over the same structure can always be combined into a single fold that generates a pair"

```
pairAlg :: Functor f => (f a → a, f b → b) → f (a, b) → (a, b)
pairAlg (fwd, bwd) = λf → (fwd (fmap fst f), bwd (fmap snd f))
```

We can encode neural network **training** as a *single fold*:

```
cata (pairAlg fwd bwd) :: Fix Layer → (Values → Values, Error → Fix Layer)
```

```
train :: Fix Layer → (Values, Values) → Fix Layer
train nn (input, desired_output) = (backward ∘ (-) desired_output ∘ forward) input
  where (forward, backward) = cata (pairAlg fwd bwd) nn
        output = forward input
        error  = output - desired_output
trainMany :: Fix Layer → List (Values, Values) → Fix Layer
trainMany = foldr (flip train)
```

Extensions

Coproducts and free monads

We can modularly encode different forms of neural networks using **coproducts**

```
data Layer k = DenseLayer Weights Biases k
             | InputLayer
```

```
data DenseLayer k = DenseLayer Weights Biases k
data InputLayer k = InputLayer

data (f :+: g) k = L (f k) | R (g k)

type Layer = (DenseLayer :+: InputLayer)
```

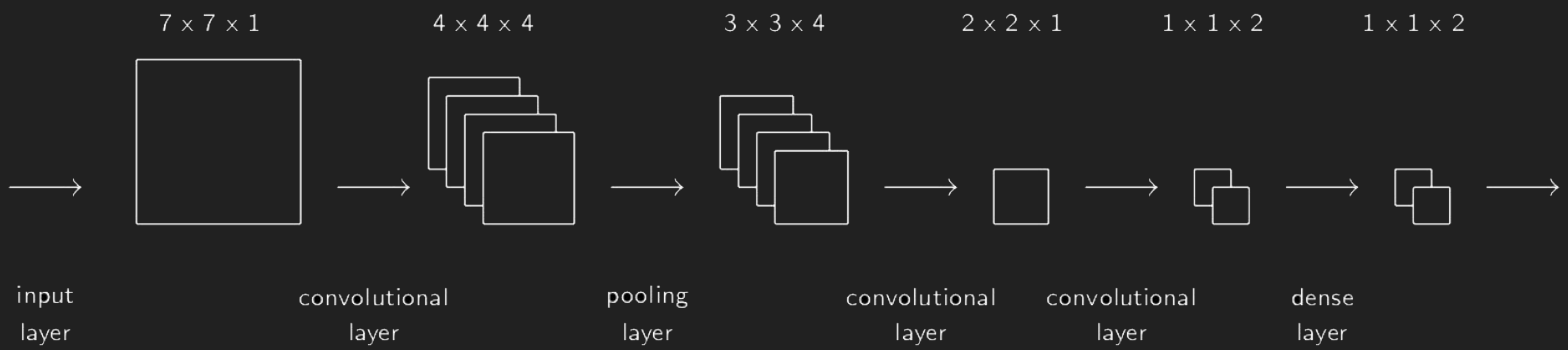
We can build neural networks extensibly using **free monads**

```
data Fix f  = In (f (Fix f))

fixNN :: Fix Layer
fixNN =
  In (DenseLayer w3 b3
   (In (DenseLayer w2 b2
     (In (DenseLayer w1 b1
       (In InputLayer))))))
```

```
data Free f a = Op (f (Free f a))
              | Pure a

freeNetwork :: Free Layer ()
freeNetwork = do
  Op (DenseLayer w3 b3 (Pure ()))
  Op (DenseLayer w2 b2 (Pure ()))
  Op (DenseLayer w1 b1 (Pure ()))
  Op (InputLayer)
```

Other neural networks

We can fold over more complex **structures**:



with more interesting patterns of **traversal**: