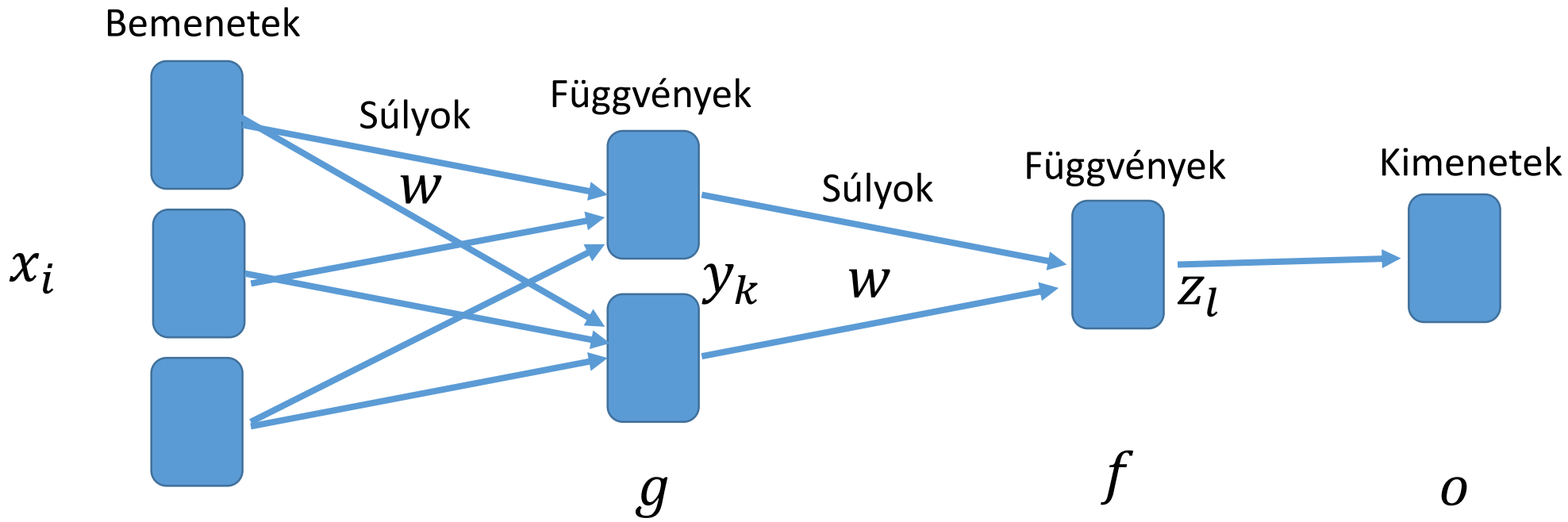
Neurális Hálók
és a
Funkcionális Programozás

Berényi Dániel
Wigner GPU Labor

Alapvető építőkövek

- Függvény kompozíció
- Automatikus Differenciálás (AD)

Neurális Háló, mint kompozíció



$$o = (f \circ g)(x_i)$$

$$net = f \circ g$$

$$y_k = g\left(\sum_i w_{ik} x_i + b_k\right) \quad z_l = f\left(\sum_k w_{kl} y_k + b_l\right)$$

Neurális Háló, mint kompozíció

- Felügyelt tanítás esetén vannak bemenet-kimenet párjaink:

$$s_q = (x_{i_q}, o_q)$$

- A háló kimenetén definiálunk egy hibafüggvényt (loss) az elvárt és kapott kimenet közé, pl.:

$$L_0(u, v) = \frac{1}{2}(u - v)^2$$

- Ha rögzítjük az elvárt kimeneteket a hibafüggvényben:

$$L(v_q) = L_0(o_q, v_q)$$

Neurális Háló, mint kompozíció

- A neurális háló tanítása a súlyok (w_ij) és biasok (b_i) beállítása
- A tanításhoz írjuk teljes hálót a hibával együtt röviden kompozícióként:

$$e(x) = (L \circ f \circ g)(x)$$

- Cél e(x) minimalizálása
- A minimumban:

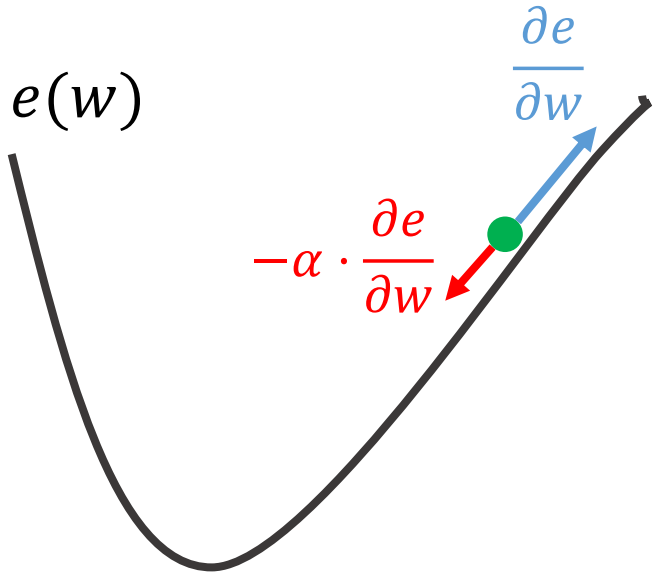
$$\frac{\partial e}{\partial w_{ij}} = 0 \quad \frac{\partial e}{\partial b_i} = 0$$

Illetve bármilyen más paraméterre a hálóban...

Neurális Háló, mint kompozíció

- A legegyszerűbb és legszéleskörűbben használt módszer a Gradiens Módszer:

$$w^{(n+1)} = w^{(n)} - \alpha \cdot \frac{\partial e}{\partial w}$$



ahol α a „tanulási ráta”

Elő kell tehát állítani a teljes háló paraméterek szerinti deriváltját

Neurális Háló, mint kompozíció

- Szükség van tehát a teljes kompozíció deriváltjára, de ismert:

$$(f \circ g)' = (f' \circ g)g'$$

- Továbbá:

$$(f \circ g \circ h)' = (f' \circ g \circ h)(g' \circ h)h'$$

Automatikus differenciálás

Ha van egy függvény kompozíciónk (vagy egy számítási gráfunk), az automatikus differenciálás segítségével kiszámolhatjuk a deriváltak numerikus értékeit, két féle módszerrel:

- Előrefelé módszer
- Hátrafelé módszer

Automatikus differenciálás

$$(f \circ g \circ h)' = (f' \circ g \circ h)(g' \circ h)h'$$

Előrefelé módszer

1. Rögzítjük a független változót, ami szerint szeretnénk deriválni pl.: x_j
2. Kiszámoljuk és lederiváljuk a legbelső lépést, pl.: $h(x_i) = x_i, \quad \frac{\partial h}{\partial x_j} = \delta_{ij}$
3. Továbbadjuk az függvény értéket és a derivált értékét $\left(h(x_i), \frac{\partial h}{\partial x_j}\right)$
4. Kiértékeljük a következő függvényt az előző értéknél $g(h(x_i))$, Kiszámoljuk a következő függvény deriváltját és beszorozzuk az előző deriválttal: $\frac{\partial g}{\partial h}\frac{\partial h}{\partial x_j}$
5. tovább adjuk a párt: $\left(g(h(x_i)), \frac{\partial g}{\partial h}\frac{\partial h}{\partial x_j}\right)$
6. Ismételjük kifelé a 4-5. lépéseket

Automatikus differenciálás

$$(f \circ g \circ h)' = (f' \circ g \circ h)(g' \circ h)h'$$

Hátrafelé módszer

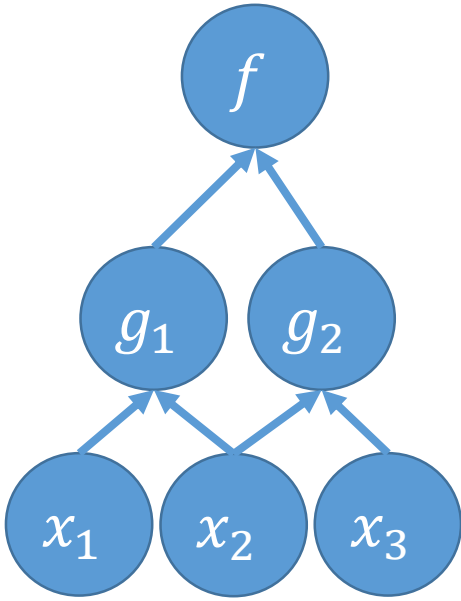
1. Kiszámoljuk előre az egyes műveleteket és letároljuk a részeredményeket
2. Rögzítjük a kimeneten (f) a függő változót, aminek a deriváltja $1 f'$
3. Kiszámoljuk ennek az utolsó függvénynek a deriváltját a bemenetek szerint, pl.:
   $f(g_1, g_2) = g_1^2 + 3g_2, \quad \frac{\partial f}{\partial g_1} = 2g_1 g_1', \quad \frac{\partial f}{\partial g_2} = 3g_2'$
   Ahol expliciten kell egy részeredmény (csak nem lineáris esetben!) oda behelyettesítjük az 1. lépésből
4. Visszük magunkkal a kapott rész kifejezéseket a megfelelő irányokba
5. Kiszámoljuk az eggyel korábbi kifejezések deriváltjait:
   $g_1 = 5h_1, \quad \frac{\partial g_1}{\partial h_1} = 5h_1' \qquad\qquad g_2 = 9 + 10h_2, \quad \frac{\partial g_2}{\partial h_2} = 10h_2'$
6. Hozzá szorozzuk a korábbi részkifejezésekhez és visszük tovább: $(2g_1 \cdot 5, 3 \cdot 10)$
7. Ismételjük kifelé

Automatikus differenciálás

Ha több változós függvényeink vannak, akkor a lánc szabály:

$$(f \circ g)' = \sum_i \frac{\partial f}{\partial g_i} \circ g_i \frac{\partial g_i}{\partial x_j}$$

Azaz, összegezni kell a bemenő élekre



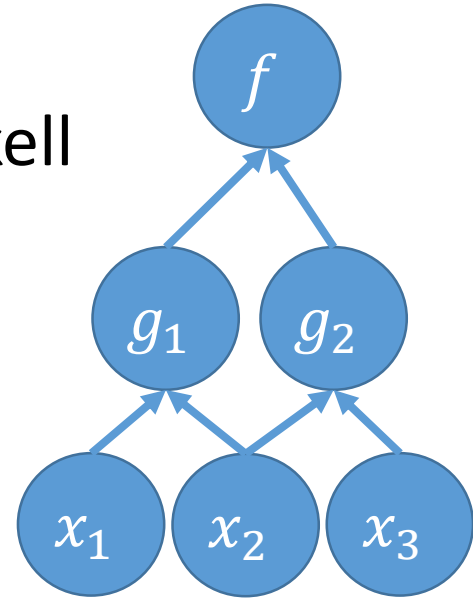
Ez iterálva kombinatorikus felrobbanáshoz vezet,
ha sok a bemenő él és az előre felé módszerrel számoljuk,
mert minden kezdőpontból nagyon sok út mentén kellene követni a deriváltakat

Automatikus differenciálás

Másképp, ha a függvényünk $\mathbb{R}^n \to \mathbb{R}^m$, és a fában N művelet van,

Akkor az előre módszernél minden változóra egyenként végig kell mennünk a fán, ami nN költségű

A hátra módszernél minden végpontból egyszer kell elindulni a fán, ezért ennek a költsége mN



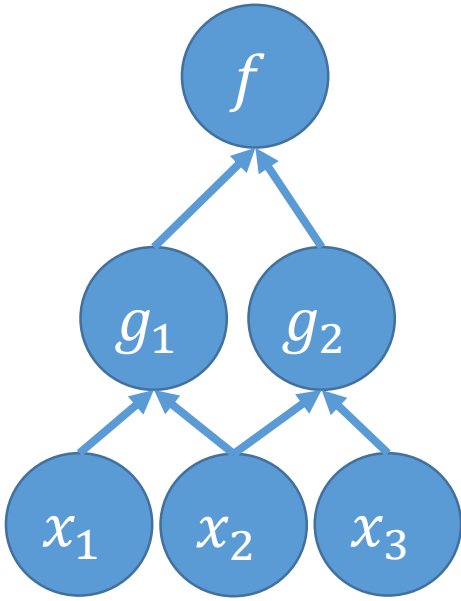
Ez a két módszer csak két szélsőség, az optimális kiértékelés NP nehéz

Automatikus differenciálás

Következmény:

Összegző, redukáló jellegű hálózatoknál a hátrafelé módszer kevesebb műveletet igényel, mint az előrefelé módszer

A kevés bemenetből egyre több és több kimenetet előállító hálózatok azonban az előrefelé módszerrel deriválhatóak könnyebben



Tanítás

A Neurális Hálók tipikusan sok adatból állítanak elő kevés adatot, ezért a hátrafelé módszerrel deriválhatóak (back propagation)

További könnyedség, hogy a legtöbb lépés lineáris, ekkor nincs szükség a függvény értékre, csak a súlyok szorzófaktoraira

A hátrafelé szállított szorzatot most δ-val fogjuk jelölni.

Tanítás

A derivált számolását a háló kimenetén kezdjük, ahol a hiba függvény (L) található:

$$f(x_i) = \sum_i w_i \, x_i + b$$

$$e' = (L \circ f)' = (L' \circ f) f'$$

$$w_i^{(n+1)} = w_i^{(n)} - \alpha \cdot \frac{\partial e}{\partial w_i} = w_i^{(n)} - \alpha \cdot \underbrace{L'\big(f(x_i)\big)}_{\delta} \cdot x_i$$

Tanítás

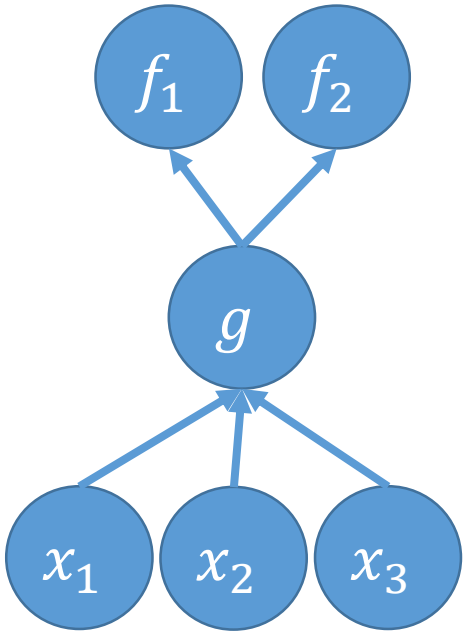
- A háló egy belső rétegében:

$$y = g(x_i) = \sum_i w_i\, x_i + b \qquad \rightarrow \qquad f_k(y) = w_k q(y)$$

$$e' = \sum_k (f_k \circ g)' = \sum_k (f_k' \circ g) \cdot g'$$

$$w_i^{(n+1)} = w_i^{(n)} - \alpha \cdot \frac{\partial e}{\partial w_i}$$

$$= w_i^{(n)} - \alpha \cdot \sum_k f_k'(g_i(x_k)) \cdot x_i$$



Tanítás

- A háló egy belső rétegében:

$$y = g(x_i) = \sum_i w_i\, x_i + b \qquad \rightarrow \qquad f_k(y) = w_k q(y)$$

$$e' = \sum_k (f_k \circ g)' = \sum_k (f_k' \circ g) \cdot g'$$

$$f_k'(y) = q'(y) w_k$$

$$\delta_k$$

$$w_i^{(n+1)} = w_i^{(n)} - \alpha \cdot \frac{\partial e}{\partial w_i}$$

$$= w_i^{(n)} - \alpha \cdot \sum_k f_k'(g_i(x_k)) \cdot x_i = w_i^{(n)} - \alpha \cdot \sum_k \delta_k w_k \cdot x_i$$

Tanítás

A paramétereket változtató gradienseket igazából fel kell összegezni a tanító halmaz elemeire, és elvileg csak azután lenne egy léptetés, ha mindegyikre kiszámoltuk a gradienst:

$s_q = (x_{i_q}, o_q)$ tanító halmaz, $q = 1 \ldots N$

$$w^{(n+1)} = w^{(n)} - \alpha \cdot \sum_{q}^{n} \frac{\partial e(x_{i_q})}{\partial w}$$

Praktikusan általában kisebb csoportokban frissítik a súlyokat (batching): $n < N$

Gyakori réteg típusok

Gyakori réteg típusok

Elemenkénti transzformáció (1 bemenet, 1 kimenet)

Leginkább a nemlinearitás, általában:

- $f(x) = \tanh(x)$
- $f(x) = |\tanh(x)|$
- $f(x) = (1 + e^{-x})^{-1}$ (szigmoid)
- $f(x) = \max(0, x)$ (Rectified Linear Unit, ReLU)
- $f(x) = \ln(1 + e^{x})$ (softplus)

Gyakori réteg típusok

Affin transzformáció:

$$f(x_i) = \sum_i w_i x_i + b$$

$$\frac{\partial f(x_i)}{\partial x_i} = w_i \qquad \frac{\partial f(x_i)}{\partial w_i} = x_i \qquad \frac{\partial f(x_i)}{\partial b} = 1$$

Gyakori réteg típusok

Teljesen összefüggő:
A két réteg minden eleme minden másikkal össze van kötve

Minden élre egyedi súlyt teszünk, azaz

Két N elemű réteg között N^2 számú súly lesz

Ez nagyon flexibilis, de az N^2 költség a gyakorlatban túl nagy

Gyakori réteg típusok

Konvolúció (igazából autokorreláció):
w kevesebb elemű, mint x, és f $|x| - |w|$ darab elemet fog tartalmazni.

$$f(x_n) = \sum_i w_i x_{n+i}$$

$$\frac{\partial f(x_n)}{\partial x_i} = w_{i-n} \qquad \frac{\partial f(x_i)}{\partial w_i} = x_{n+i}$$

Gyakori réteg típusok

Redukció (pooling, subsampling):

Lényegében bármilyen $\mathbb{R}^n \to \mathbb{R}$ deriválható fv., a lényeg: nincs átfedés a bemenetek között (szemben a konvolúcióval).

$$y = f(x_1, x_2, \dots x_n)$$

Példák:

átlag: $f(x) = \frac{1}{n}\sum_i^n x_i\,, \qquad \frac{\partial f}{\partial x_i} = \frac{1}{\mathrm{n}}$

max: $f(x) = \max(x_i)\,, \qquad \frac{\partial f}{\partial x_j} = \begin{cases} 1, & x_j = \max(x_i) \\ 0, & else \end{cases}$

A fordított is létezik, de nehezebb: dekonvolúció

Gyakori réteg típusok

Más redukció, kifejezetten a hibafüggvényre: Cross-entropy

$$C = -\frac{1}{n}\sum_{x} o \ln y + (1-o)\ln(1-y)$$

ahol o az elvárt kimenet, y az aktuális kimenet, x-ek a bemenetek,

$$y = f\left(\sum_{i} w_i x_i + b\right)$$

Tulajdonságok: $C > 0$, ha belül minden változó [0, 1]-beli
Ennek a deriváltja nem arányos f'-el, ezért nem lassul le a tanulás kis hibáknál.

Gyakori réteg típusok

Redukciós transzformáció elemsokaságon

Példa: Softmax $\mathbb{R}^n \to \mathbb{R}^n$

$$f(x_1, \ldots, x_n)_i = \frac{e^{x_i}}{\sum_k e^{x_k}}$$

Ez egy normalizált összeg, ahol a végén $\sum_i f_i = 1$

Tipikusan a klasszifikáló hálók kimenetén található

Bonyolultabb hálózatok

- Az eddigi hálók irányított aciklikus gráfokkal reprezentálhatóak

- Ez izomorf egy névfeloldásos számítási fával, mint egy lambda kalkulus

Rekurrens hálózatok

Visszakötés, visszacsatolás a saját/korábbi csúcsokhoz

LSTM

Manapság a legnépszerűbb RNN: Long short-term Memory (LSTM) (link)

- Könnyű tanítani, 4 súly f:felejtés, i:input, o:output, c:cell (memória)

$$C_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] + b_f\right) * C_{t-1} + \sigma(W_i \cdot [h_{t-1}, x_t] + b_i) * \tanh(W_C \cdot [h_{t-1}, x_t] + b_C)$$

$$h_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o) * \tanh(C_t)$$

*: pontonkénti szorzás Forrás: colah.github.io

LSTM tanítás

Az LSTM és néhány más rekurrens háló úgy tanítható, hogy valamennyi lépésig kitekerjük az ismétlést:

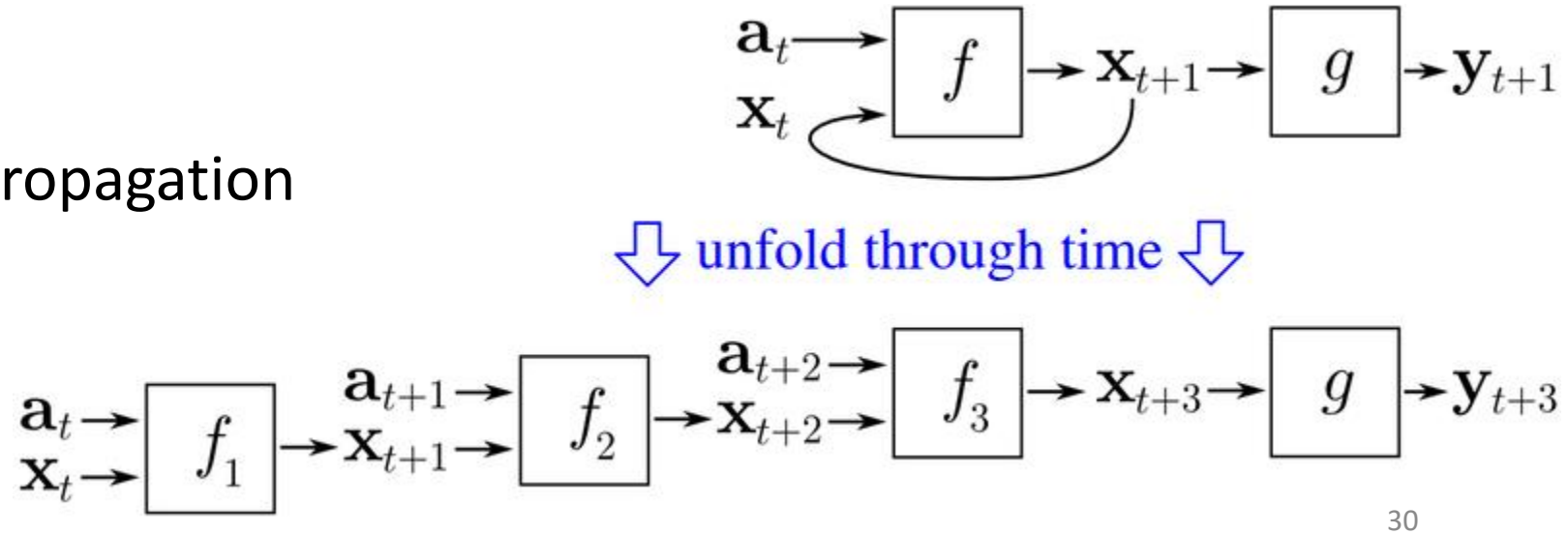
Ezek után olyan hosszú
bemenet-kimenet pár sorozat kell,

amennyi lépést kitekertünk

+egy kezdő tipp x0-ra

Innen már a szokásos backpropagation

használható



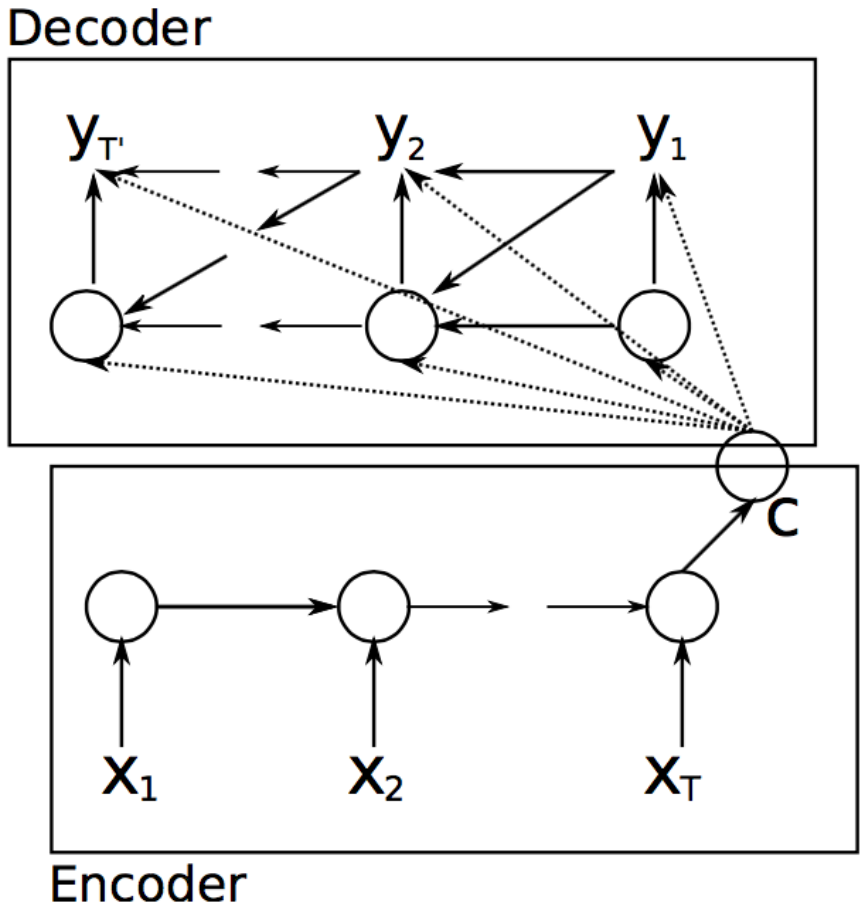
Encoder-Decoder RNN-ek

Eredeti cikk: K. Cho et al

- Encoder: egy változó hosszú inputot egy fix hosszú vektorba kódol

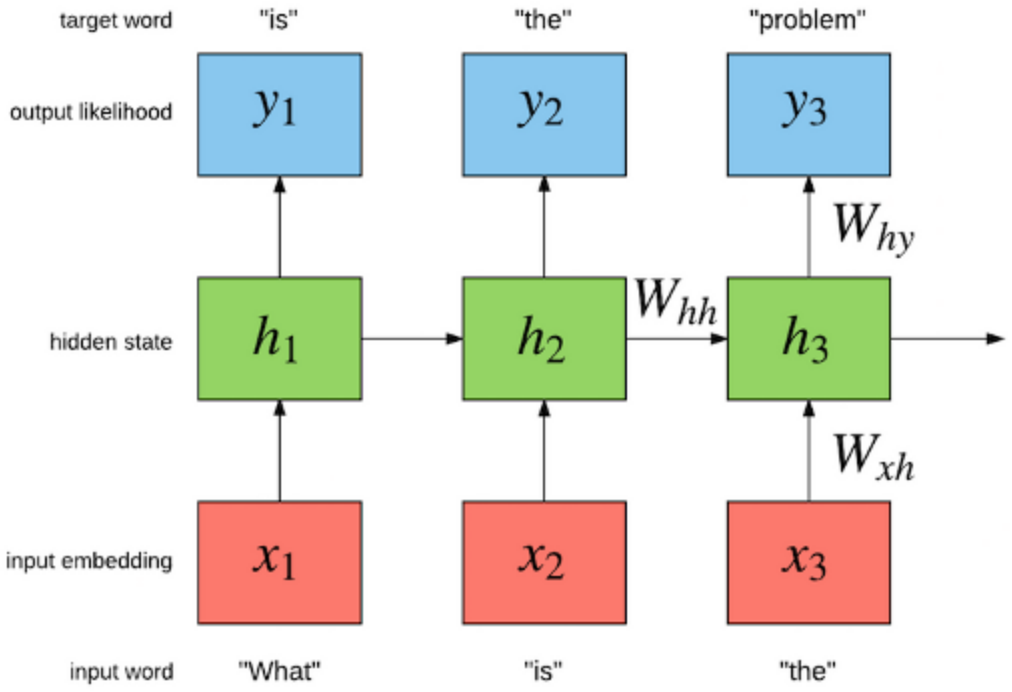
- Decoder: egy fix hosszú inputot egy változó hosszú outputtá alakít

Az adatok végét speciális értékek jelzik mindkét esetben



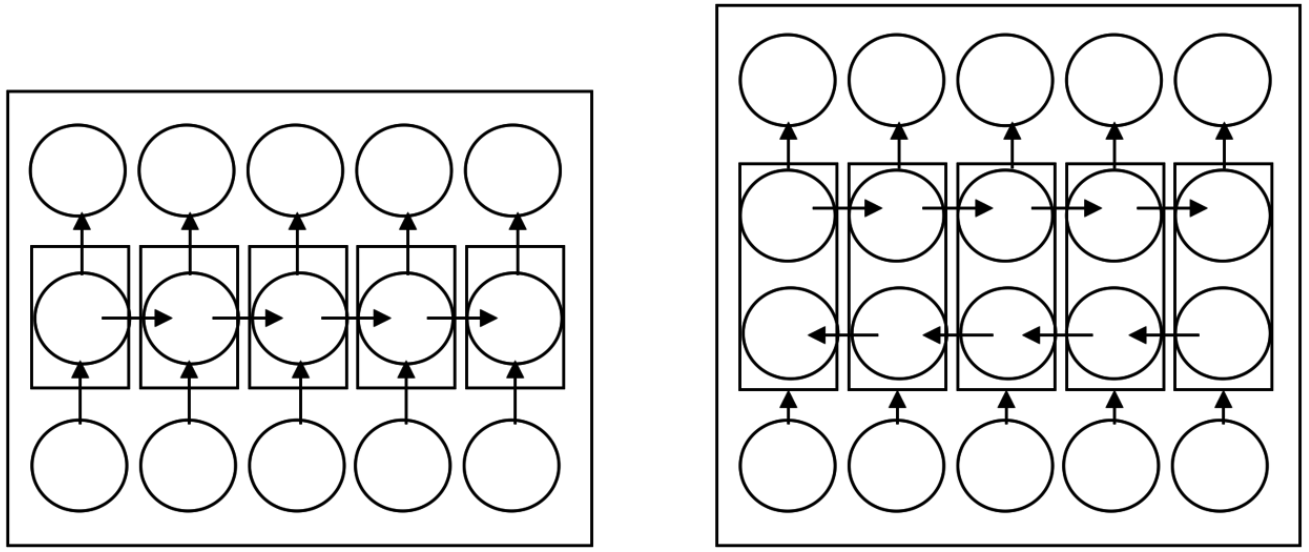
Általános adatfolyam transzformátorok

- Ha egyszerre történik az encode és decode lépés, az lényegében egy általános RNN, általános transzformációról beszélhetünk



Kétirányú transzformátorok

- Ha a tradicionális RNN-el szemben ellenkező irányba is van csatolás
- Tanítás a két irányban függetlenül



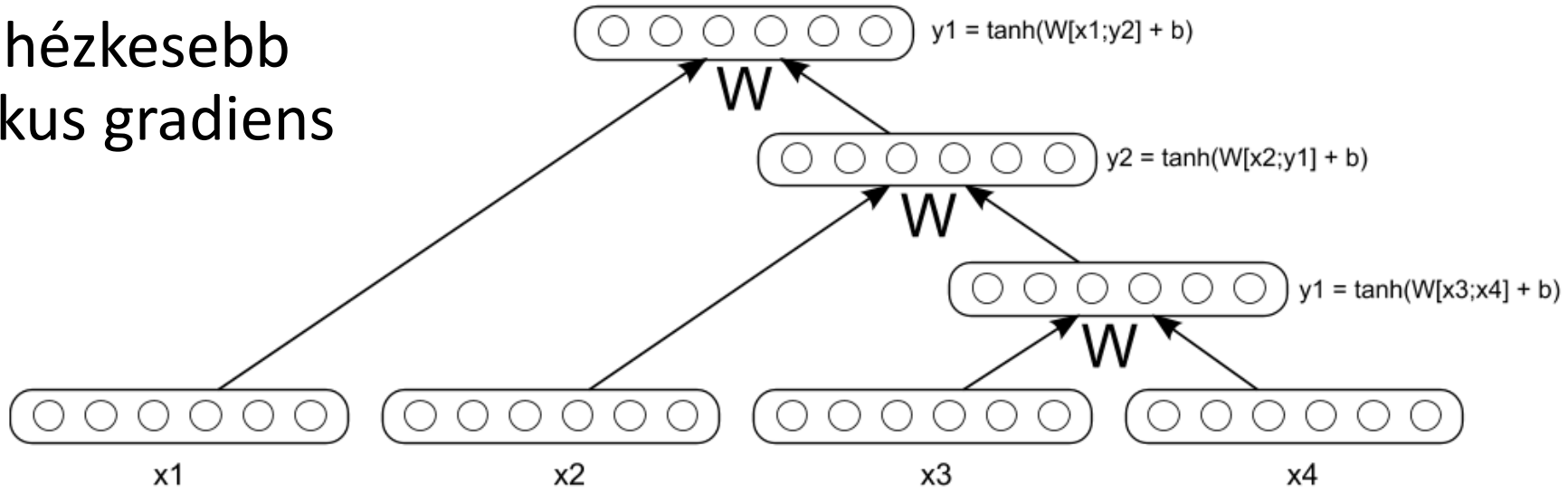
Structure overview
(a) unidirectional RNN
(b) bidirectional RNN

RNN alkalmazási területek

- Természetes nyelvek feldolgozása (fordítás, beszéd felismerés)
- Kézírás felismerés
- Idősorok jóslása (akár zene)
- Kémiai szerkezet jóslás

Rekurzív Neurális Hálók (Nem rekurrens!)

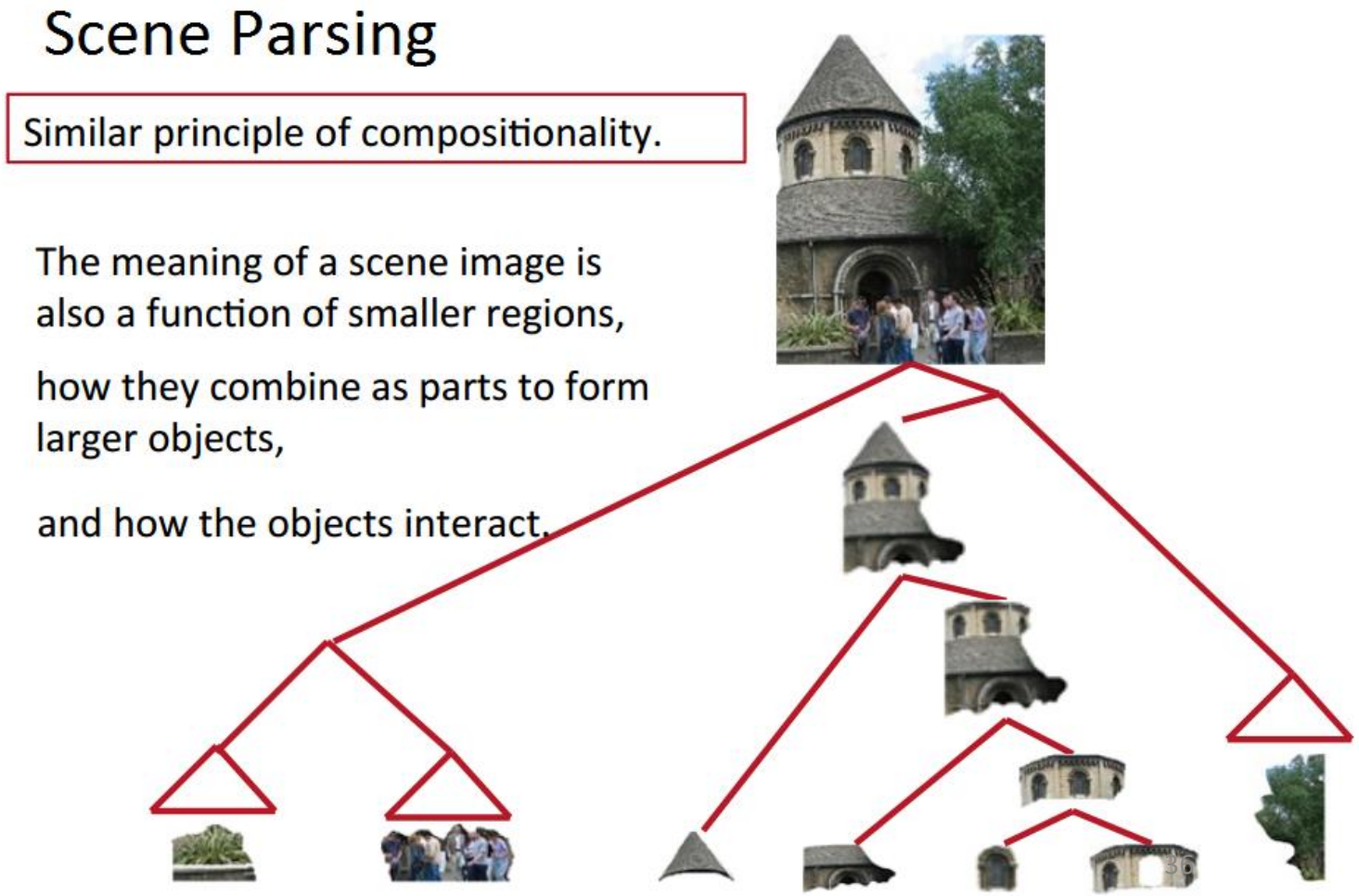
- Itt egy fa struktúrában minden elágazásál ugyan azokat a súlyokat használják
- A tanítás nehézkesebb (sztochasztikus gradiens módszer)



Rekurzív Neurális Hálók (Nem rekurrens!)

Alkalmazások:

- Mondatelemzés
  - Parafrázis detektálás
- Kép értelmezése



NN programozási rendszerek

- Caffe (UC Berkeley, C++, python)
- TensorFlow (Google, C++, python)
- Theano (Université de Montréal, python)
- Torch (C++, Lua)

Funkcionális Programozás

Mi köze a neurális hálóknak
a funkcionális programozáshoz?

Funkcionális Programozás

1.: A hálók általánosan egy irányított gráffal reprezentálhatóak

Ha a rekurrens részeket nem tekintjük, akkor irányított aciklikus gráffal (Directed Acyclic Graph, DAG)

Ezek izomorfak egy névfeloldásos számítási fával, ami izomorf a lambda kalkulussal, ami mint korábban láttuk a Cartesian Closed Category-k természetes nyelve

Ha van ciklikus rész, az egy rekurzióval helyettesíthető

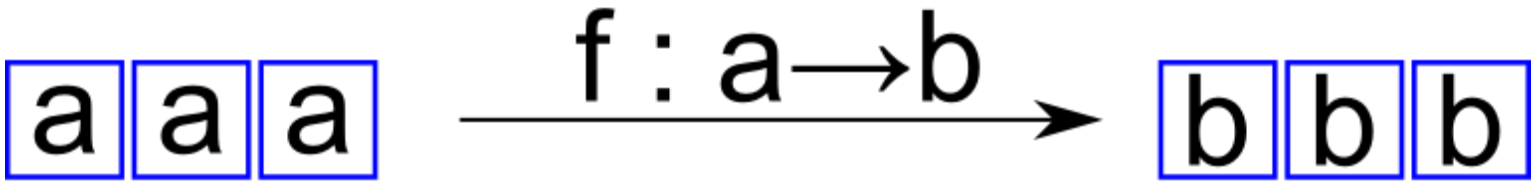
Funkcionális Programozás

2.: A hálók alapvető építőköveiben felismerhetjük az ismert funkcionális programozási magasabb rendű műveleteket

Funkcionális Programozás

Elemenkénti nemlinearitás, vagy más függvény hattatása egy Funktor struktúrára (s):

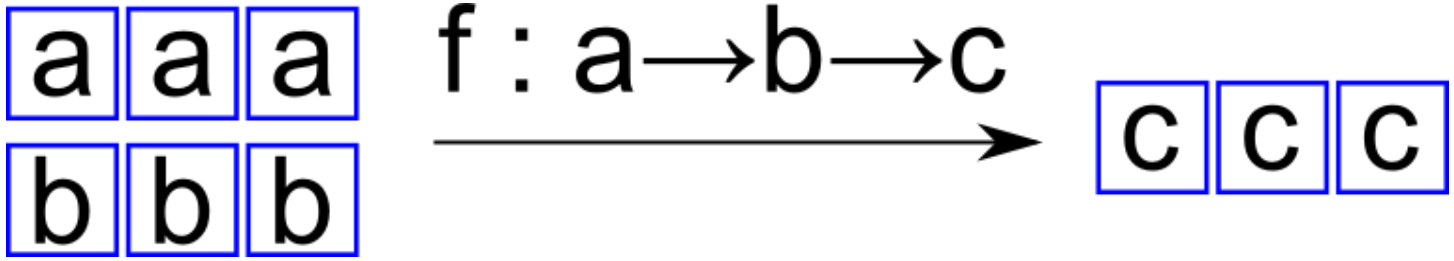
```
map :: (a->b) -> s a -> s b
```



Funkcionális Programozás

Elemenkénti összefésülés, egyesítés:

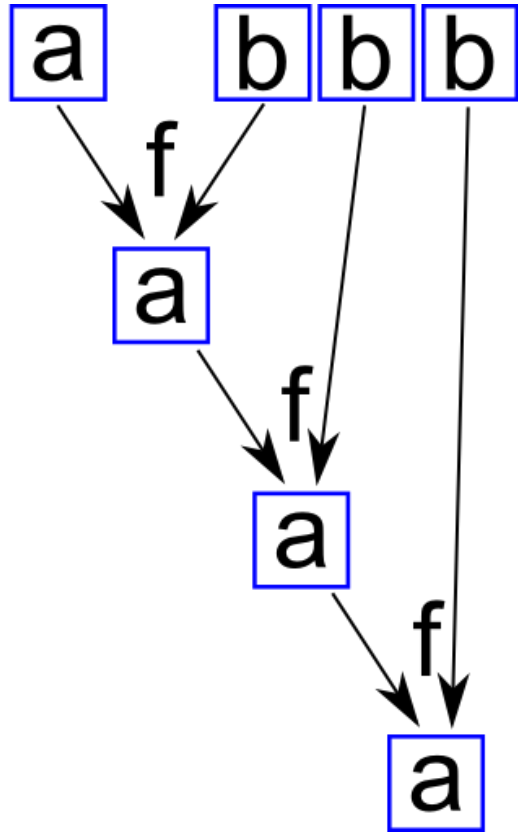
```
zip :: (a->b->c) -> s a -> s b -> s c
```



Funkcionális Programozás

Elemsokaság redukálása egy bináris művelettel:

$$f : a \rightarrow b \rightarrow a$$



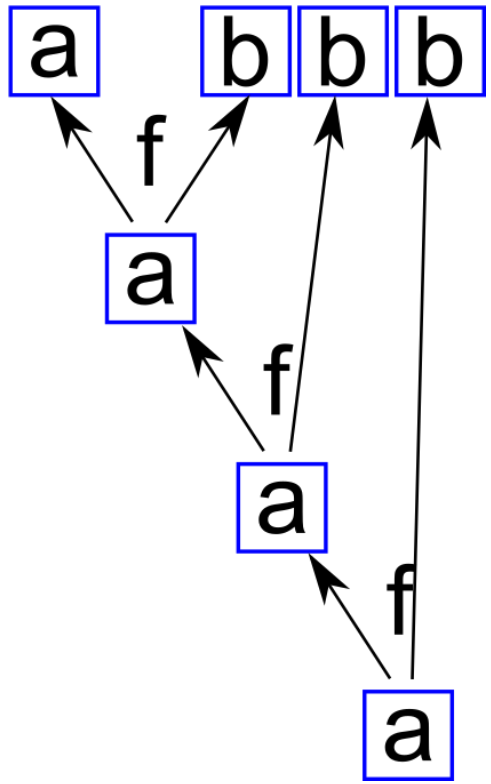

```
foldl :: (a->b->a) -> a -> s b -> a

foldr :: (b->a->a) -> a -> s b -> a
```

Funkcionális Programozás

Egy elemből elemsokaság előállítása egy művelettel:

$$f : a \rightarrow (a, b)$$

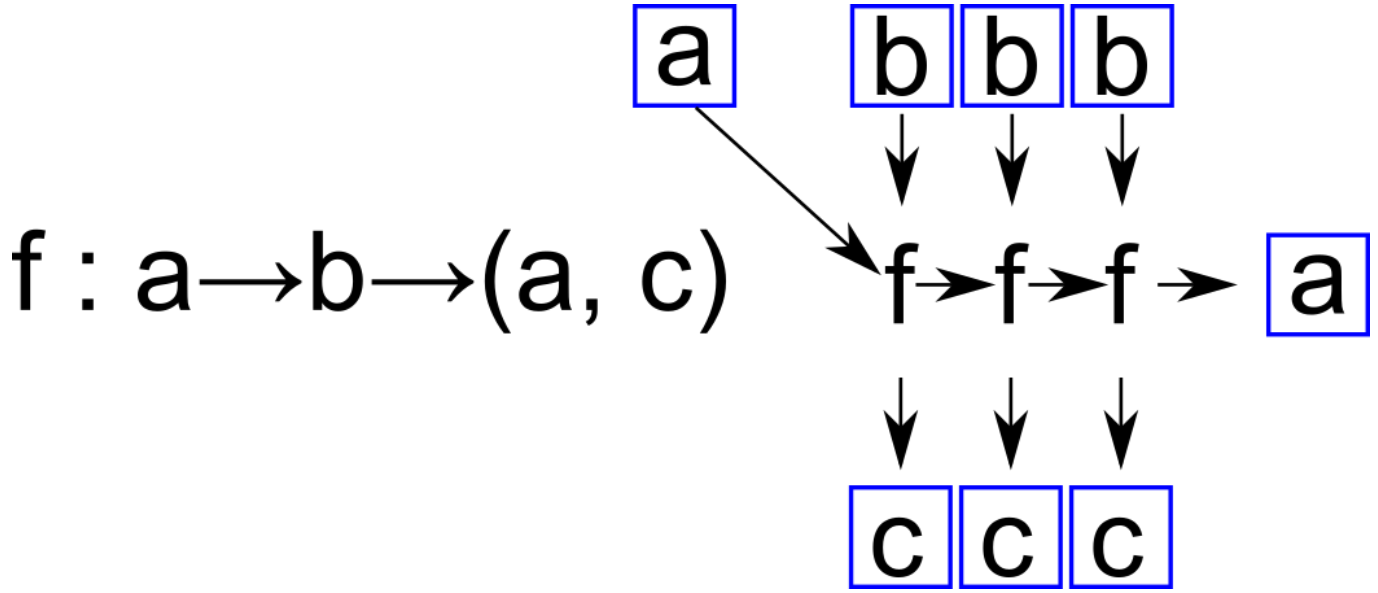



```
unfoldl :: (a->(a, b)) -> a -> s b

unfoldr :: (b->(b, a)) -> a -> s b
```

Funkcionális Programozás

Elemsokaságból redukció és
transzformáció egyben
(egy map és egy fold
lépésenkénti összefésülése)




```
mapaccuml :: (a->b->(a, c))-> a -> s b -> (a, s c)
```

Funkcionális Programozás

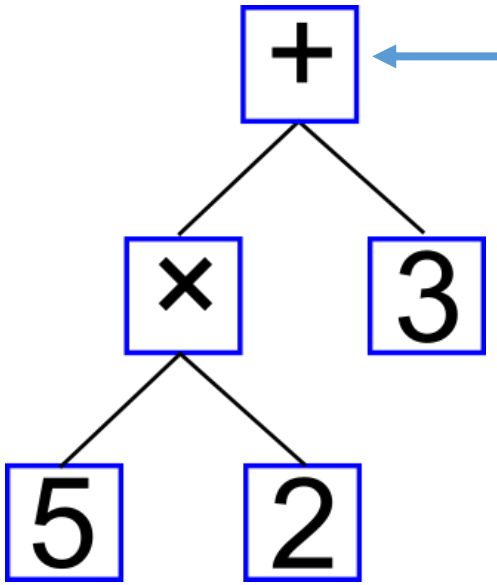
Rekurzív struktúrák redukálása / alulról felfelé transzformációja:

Katamorfizmus:

```
cata :: (f a -> a) -> Fix f -> a
cata alg tree = (alg . (map cata alg) . unfix) tree
```

Funkcionális Programozás

```
cata :: (f a -> a) -> Fix f -> a
cata alg tree = (alg . (map cata alg) . unfix) tree
```

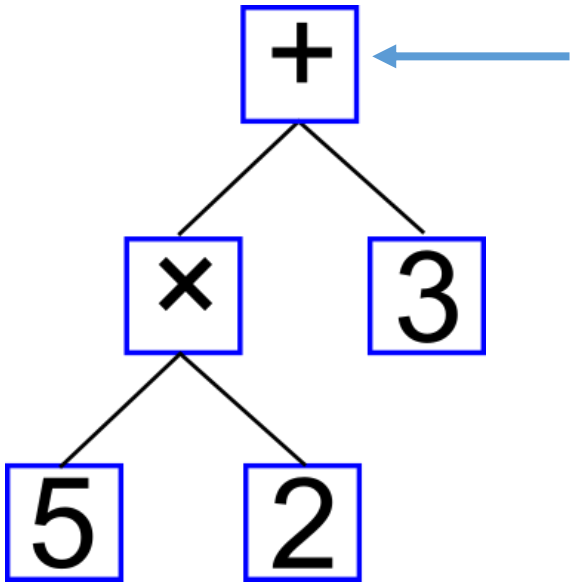


Típusa: `Fix f`
ahol `f a = Const Int | Add a a | Mul a a`

Funkcionális Programozás

```
cata :: (f a -> a) -> Fix f -> a
cata alg tree = (alg . (map cata alg) . unfix) tree
```

```
f a = Const Int | Add a a | Mul a a
```

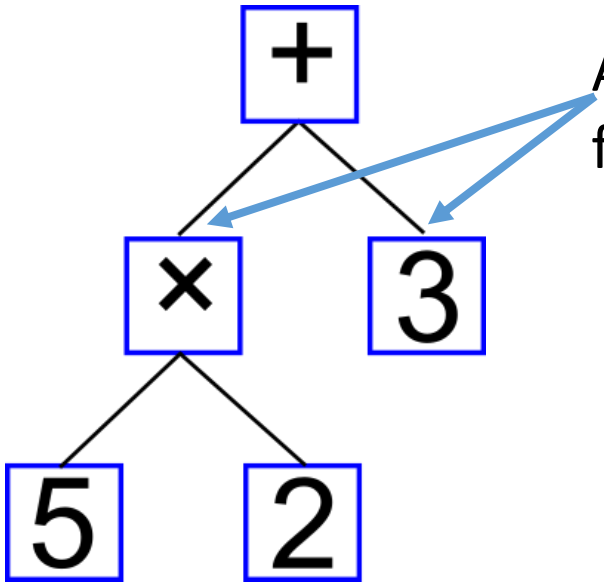


Az unfix után a típusa: `f (Fix f)`

Funkcionális Programozás

```
cata :: (f a -> a) -> Fix f -> a
cata alg tree = (alg . (map cata alg) . unfix) tree
```

```
f a = Const Int | Add a a | Mul a a
```

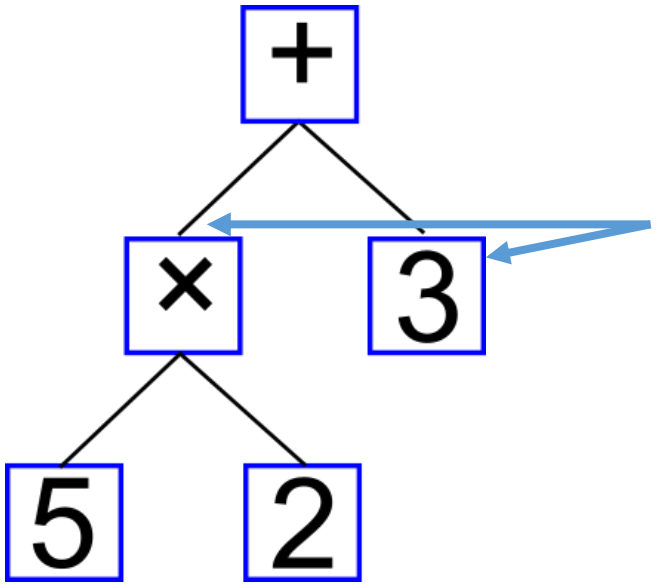


A map mind két gyerekre hattatja a (cata alg) függvényt

Funkcionális Programozás

```
cata :: (f a -> a) -> Fix f -> a
cata alg tree = (alg . (map cata alg) . unfix) tree
```

```
f a = Const Int | Add a a | Mul a a
```

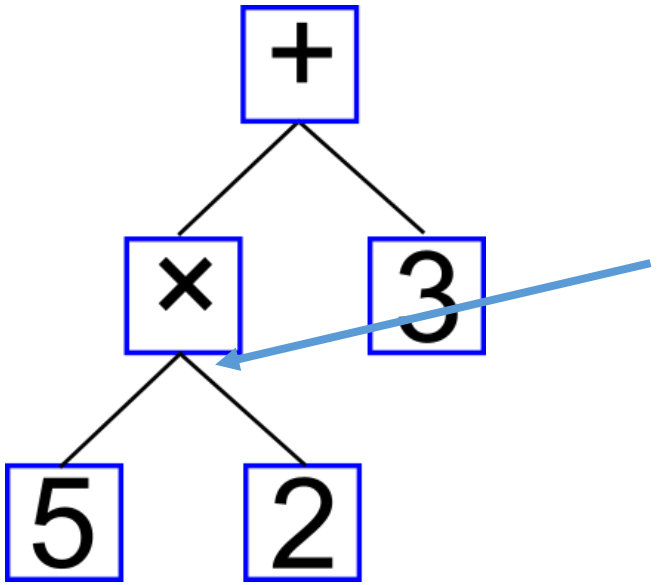


Újabb unfix, felfedi a belső típust...

Funkcionális Programozás

```
cata :: (f a -> a) -> Fix f -> a
cata alg tree = (alg . (map cata alg) . unfix) tree
```

```
f a = Const Int | Add a a | Mul a a
```

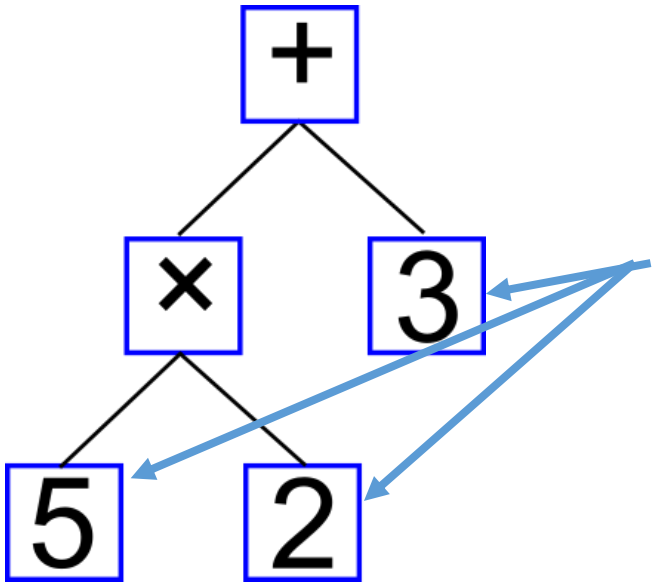


Ezen az ágon még egy map fog történni... Majd utána unfix...

Funkcionális Programozás

```
cata :: (f a -> a) -> Fix f -> a
cata alg tree = (alg . (map cata alg) . unfix) tree
```

```
f a = Const Int | Add a a | Mul a a
```

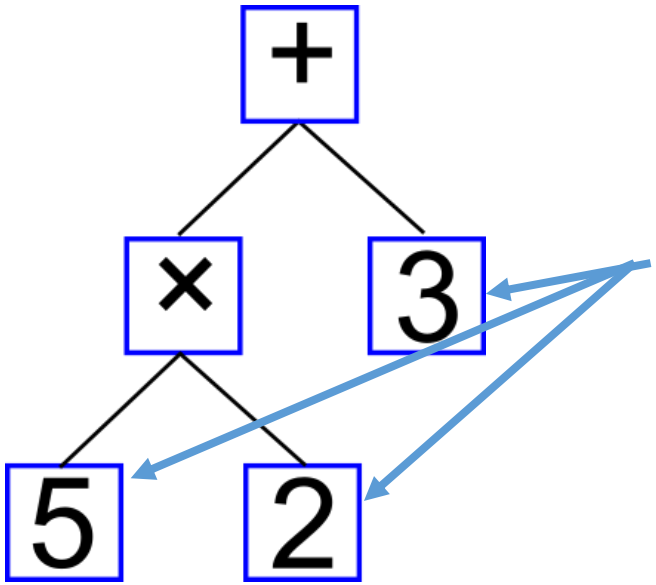


A végső konstans ágakban a map nem csinál semmit, mert nincs gyerek...

Funkcionális Programozás

```
cata :: (f a -> a) -> Fix f -> a
cata alg tree = (alg . (map cata alg) . unfix) tree
```

```
f a = Const Int | Add a a | Mul a a
```

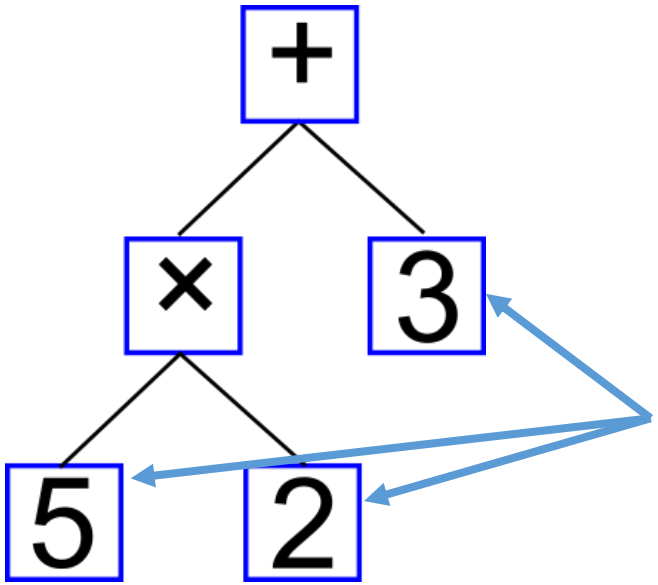


Ekkor jutunk csak el oda, hogy hattassuk alg-ot,
az *algebrát*

Funkcionális Programozás

```
cata :: (f a -> a) -> Fix f -> a
cata alg tree = (alg . (map cata alg) . unfix) tree
```

```
f a = Const Int | Add a a | Mul a a
```



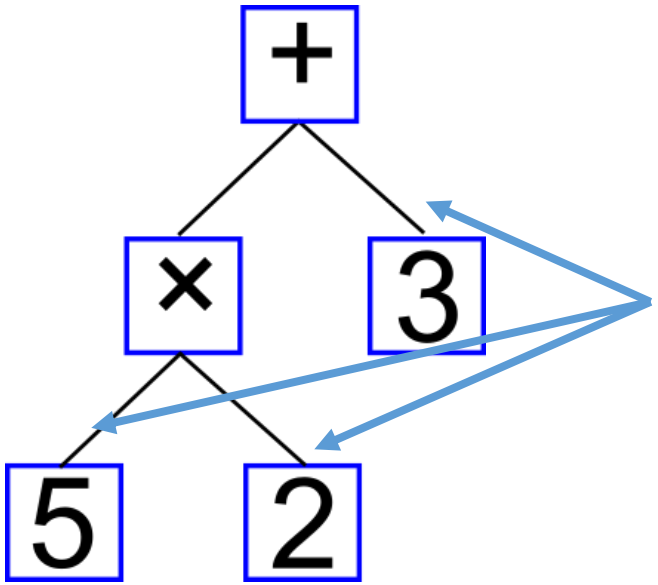
Legyen most az algebra egy kiértékelő algebra,
ami a következőt csinálja:

Const esetben visszaadja a tárolt Int-et
Add esetben visszaadja a két ág összegét
Mul esetben visszaadja a két ág szorzatát

Funkcionális Programozás

```
cata :: (f a -> a) -> Fix f -> a
cata alg tree = (alg . (map cata alg) . unfix) tree
```

```
f a = Const Int | Add a a | Mul a a
```

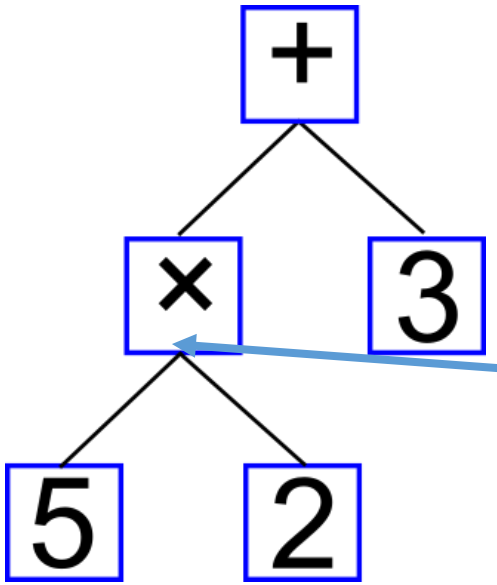


`alg` tehát először a konstansokat értékeli ki, és visszaadja az előző függvényhívásba
A visszaadott érték `a=Int`, ez a *carrier type*

Funkcionális Programozás

```
cata :: (f a -> a) -> Fix f -> a
cata alg tree = (alg . (map cata alg) . unfix) tree
```

```
f a = Const Int | Add a a | Mul a a
```



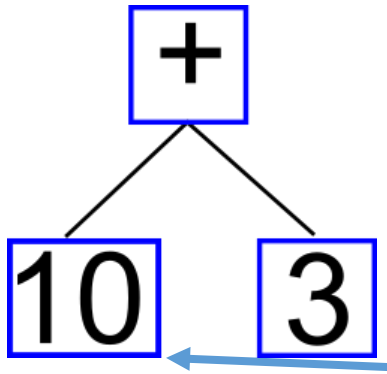
A szorzásnál a `map cata alg` eredménye egy `f Int`, mégpedig `Mul Int Int` ággal

Tehát ezt az `alg` megfelelő ága le tudja kezelni és kiszámolja a szorzatot

Funkcionális Programozás

```
cata :: (f a -> a) -> Fix f -> a
cata alg tree = (alg . (map cata alg) . unfix) tree
```

```
f a = Const Int | Add a a | Mul a a
```



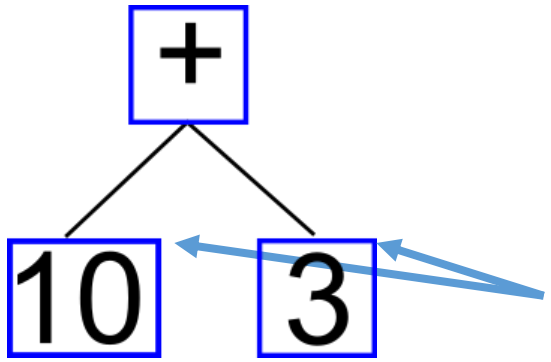
A szorzásnál a `map cata alg` eredménye egy `f Int`, mégpedig `Mul Int Int` ággal

Tehát ezt az `alg` megfelelő ága le tudja kezelni és kiszámolja a szorzatot

Funkcionális Programozás

```
cata :: (f a -> a) -> Fix f -> a
cata alg tree = (alg . (map cata alg) . unfix) tree
```

```
f a = Const Int | Add a a | Mul a a
```



Ekkor a legkülső cata híváshoz térünk vissza
Ahol az `alg Add Int Int` ága összeadja a 10-et és 3-at.

Funkcionális Programozás

```
cata :: (f a -> a) -> Fix f -> a
cata alg tree = (alg . (map cata alg) . unfix) tree
```

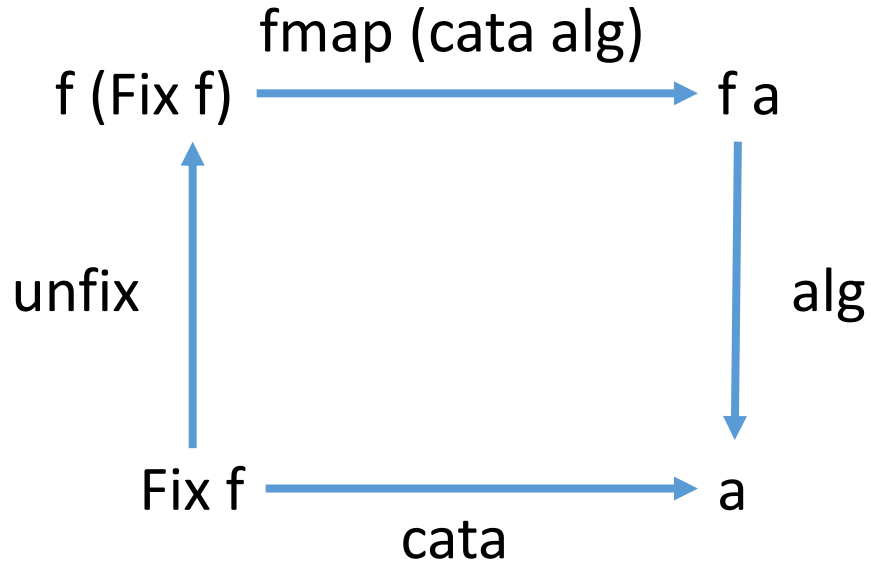
```
f a = Const Int | Add a a | Mul a a
```

Ekkor a legkülső cata híváshoz térünk vissza
Ahol az `alg Add Int Int` ága összeadja a
10-et és 3-at.

Funkcionális Programozás

```
cata :: (f a -> a) -> Fix f -> a
```

A katamorfizmus direktben kategóriaelméletből jön ki (linkek),
ahol rekurzív típusokra belátható az alábbi diagramm kommutálása:



Funkcionális Programozás

A katamorfizmusok speciális esetei a fold-ok,
hiszen a listák (vektorok) rekurzív típusok:

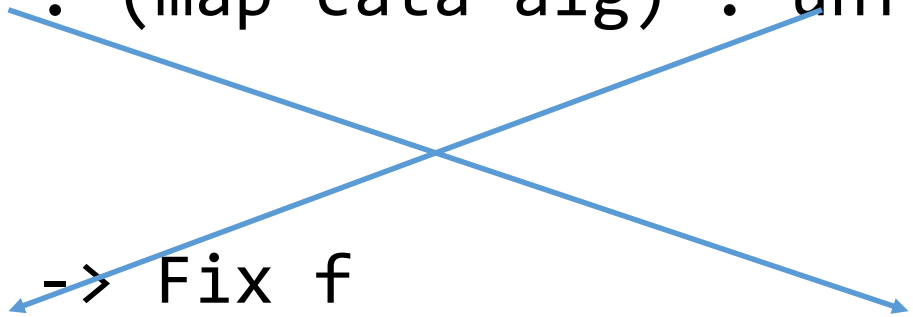
```
List a r = [a] = Empty | Node a r
```

ahol `a` a tárolt adat típusa, `r` a rekurzív pozíció.

Funkcionális Programozás

A katamorfizmusok duálisai az anamorfizmusok,
Amelyek lebontás helyett felépítenek, vagy felülről lefelé járnak be.
Ezek spec esetei az unfoldok.

```
cata :: (f a -> a) -> Fix f -> a
cata alg tree = (alg . (map cata alg) . unfix) tree
```



```
ana :: (a -> a) -> a -> Fix f
ana coalg tree = (fix . (map ana coalg) . coalg) seed
```

Funkcionális Programozás

A funkcionális építőkövek és réteg típusok megfeleltetése:

Háló réteg típus	Funkcionális primitív
Elemenkénti függvény hattatás	map
Kódoló RNN	fold
Dekódoló RNN	unfold
Általános RNN	mapaccum
Kétirányú RNN	Zip f (mapaccuml ...) (mapaccumr ...)
Konvolúció első szomszéddal	Zip f xs (tail xs)
Rekurzív NN	cata

Funkcionális Programozás

3.: A FP magasabb rendű függvényei differenciálható függvényekre differenciálható függvényeket eredményeznek

Ezért ha ezeket komponáljuk, akkor az egész kifejezés deriválható lesz.

A neurális hálók optimalizálása (tanítása) ezért lehetséges

Funkcionális Programozás

3.: A FP magasabb rendű függvényei differenciálható függvényekre differenciálható függvényeket eredményeznek

Ez onnan látható, hogy a magasabbrendű függvények csak a szerkezetet mondják meg (gráfként nézve a topológiát), a konkrét függvényt mindig a felhasználó adja meg.

Ha a megadott függvény differenciálható, akkor a magasabbrendű függvény is az.

Funkcionális Programozás

4.: Mivel a hálók függvények kompozíciói, ezért tárgyalhatóak kategória elméleten belül, és azonosságok vezethetőek le számos speciális esetben.

Ezek a fúziós összefüggések lehetővé teszik egyes lépések összeolvasztását, egyszerűsítését.

```
map f . map g = map (f.g)
zip f (map g u) (map h v) = zip (\x, y->f (g x) (h y)) u v
fold f z (map g u) = fold (\x, y -> f x (g y)) u
cata alg1 . cata alg2 = cata (alg1 . alg2)
```

Összefoglalás

- A neurális hálók lényegében differenciálható függvények kompozíciói
- A hálók rétegei magasabb rendű függvényekként is tekinthetőek
- Ezek fajtái kategória elméletből motiváltak és innen következnek a műveleti tulajdonságaik is
- A műveleti tulajdonságok következményeként hatékony eljárások léteznek a deriválásra (ami a tanítás alapvető eleme), valamint az egyszerűsítésre