Ist das ein Graph oder kann das weg? Funktionales Deep Learning in Haskell

Raoul Schlotterbeck

Created: 2023-03-16 Thu 16:00

Compiling to Categories

"As so often, I find that talks that I'm giving are basically explaining what Conal Elliot and Ed Kmett have done several years ago."

- Simon Peyton Jones

Compiling to Categories

CONAL ELLIOTT, Target, USA

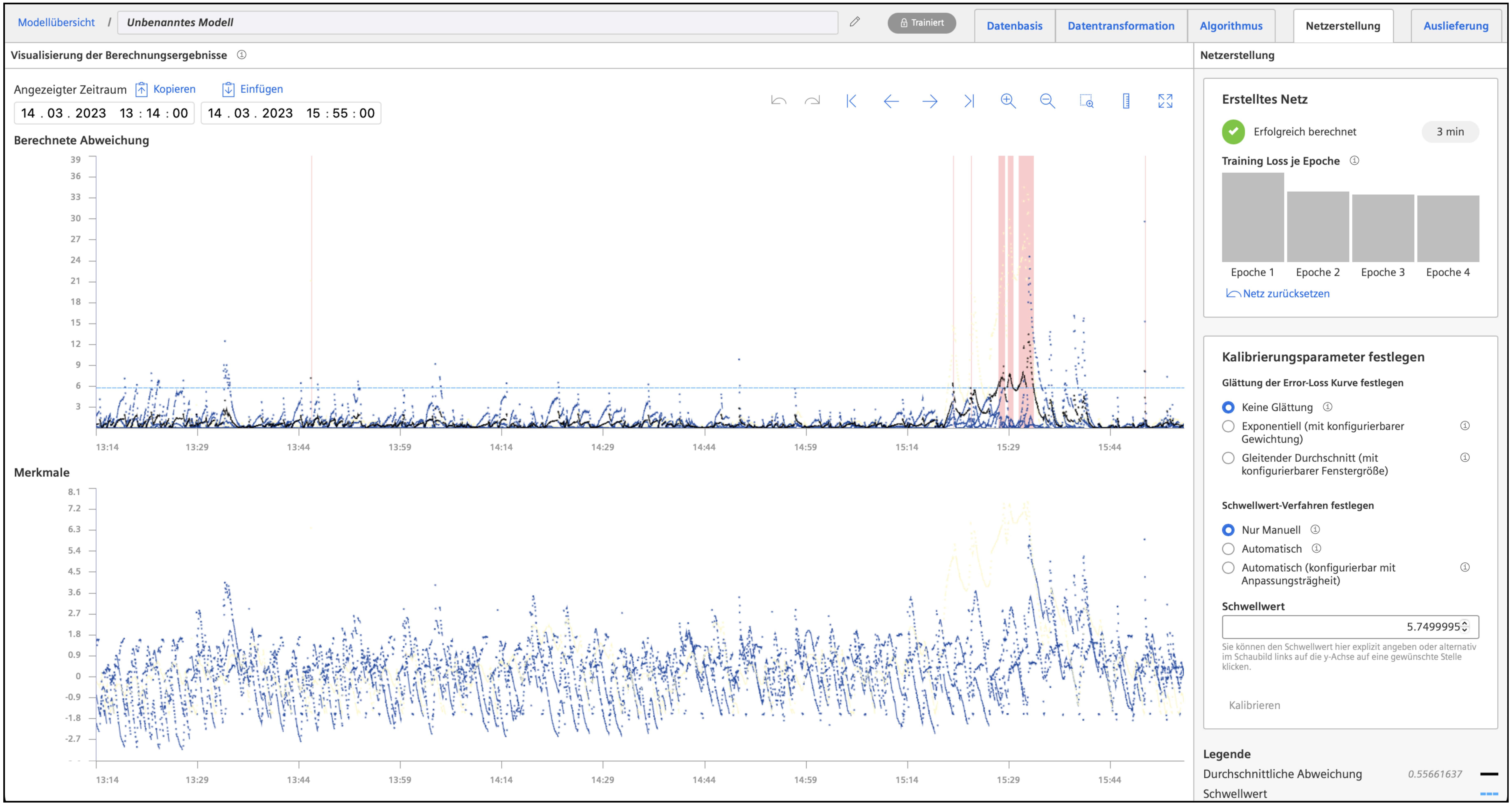
It is well-known that the simply typed lambda-calculus is modeled by any cartesian closed category (CCC). This correspondence suggests giving typed functional programs a variety of interpretations, each corresponding to a different category. A convenient way to realize this idea is as a collection of meaning-preserving transformations added to an existing compiler, such as GHC for Haskell. This paper describes such an implementation and demonstrates its use for a variety of interpretations including hardware circuits, automatic differentiation, incremental computation, and interval analysis. Each such interpretation is a category easily defined in Haskell (outside of the compiler). The general technique appears to provide a compelling alternative to deeply embedded domain-specific languages.

27

CCS Concepts: • **Theory of computation** → *Lambda calculus*; • **Software and its engineering** → *Functional languages*; *Compilers*;

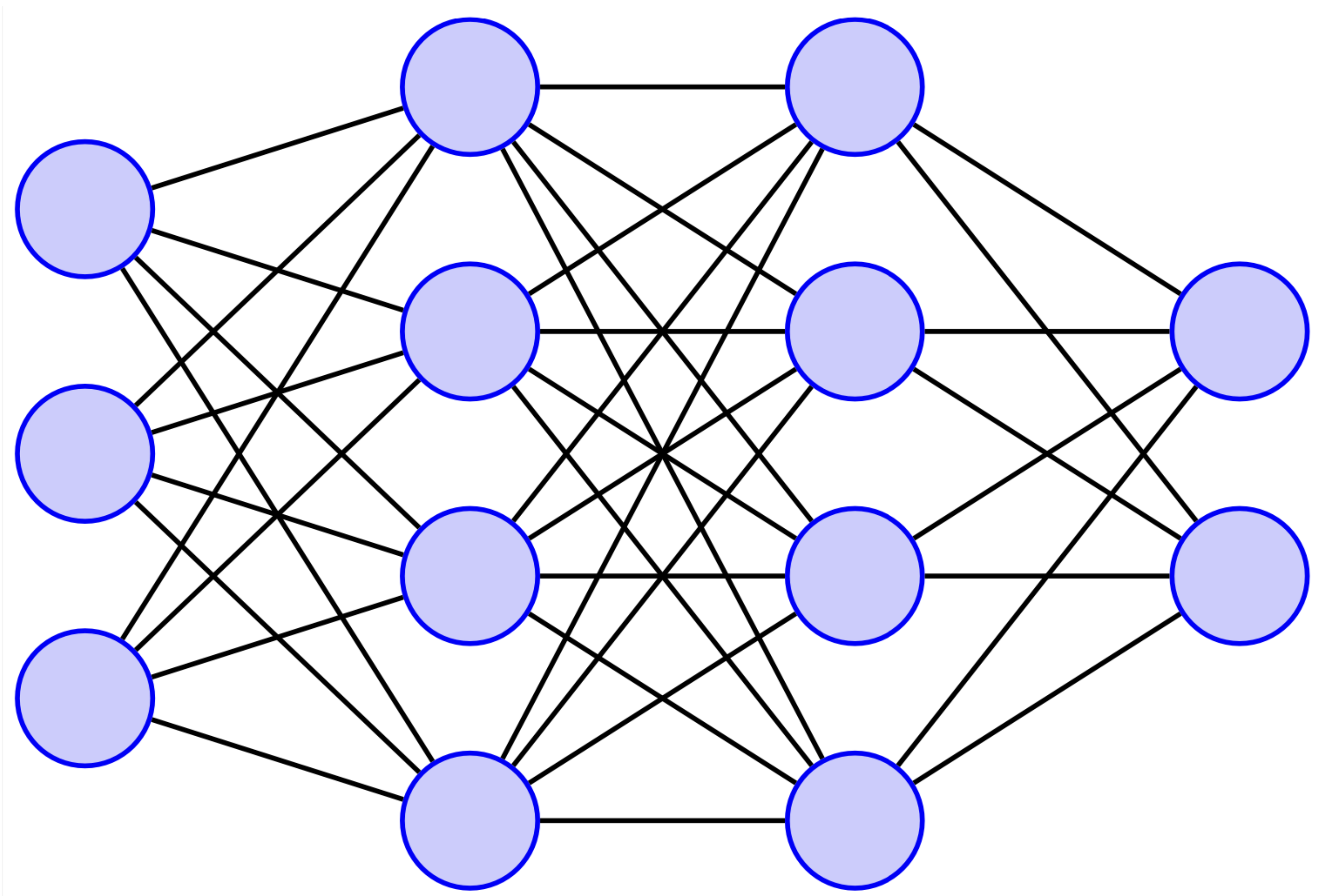
Additional Key Words and Phrases: category theory, compile-time optimization, domain-specific languages

Siemens Anomaly App



Neuronale Netze sind doch nur Funktionen

```
neuralNet wL bL ... w1 b1 =
  layer wL bL
  . ...
  . layer w1 b1

layer weightMatrix biasVector =
  fmap activationFunction
  . vectorAddition biasVector
  . matrixVectorProduct weightMatrix
```



$\Rightarrow$ Komposition purer Funktionen

Deep learning community: hold my beer

"[...] layers (die im modernen maschinellen Lernen als **zustandsbehaftete** Funktionen mit **impliziten Parametern** verstanden werden sollten) werden typischerweise als **Python-Klassen** dargestellt, deren Konstruktoren ihre Parameter **erzeugen und initialisieren** [...]"

übersetzt aus: Paszke, Adam, et al. "Pytorch: An imperative style, high-performance deep learning library." Advances in neural information processing systems 32 (2019)

Trainingsalgorithmus

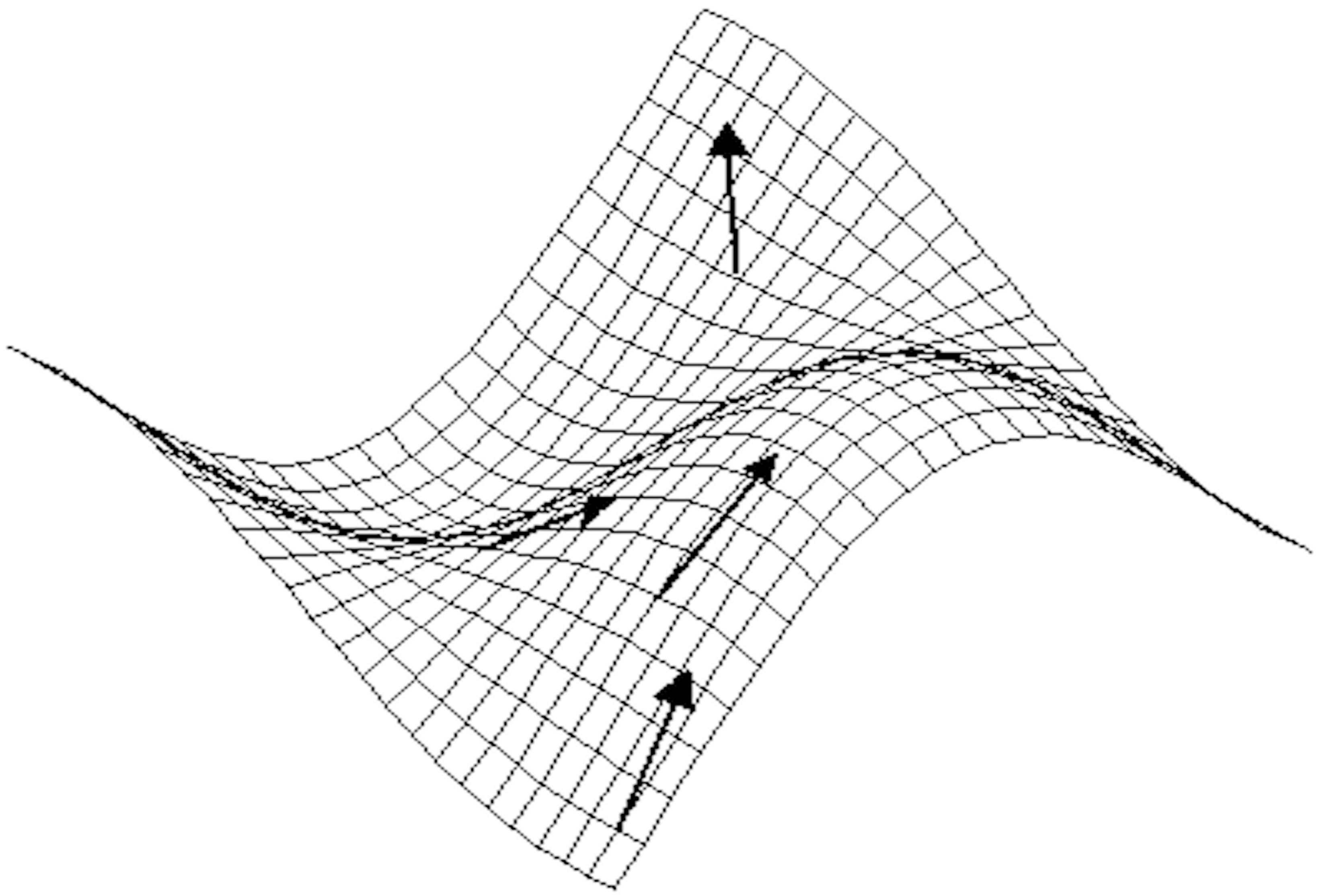
- Finde Parameter, für die das NN auf gegebenem Trainingsdatensatz möglichst gute Ergebnisse liefert
- Güte durch skalarwertige Fehlerfunktion beurteilt

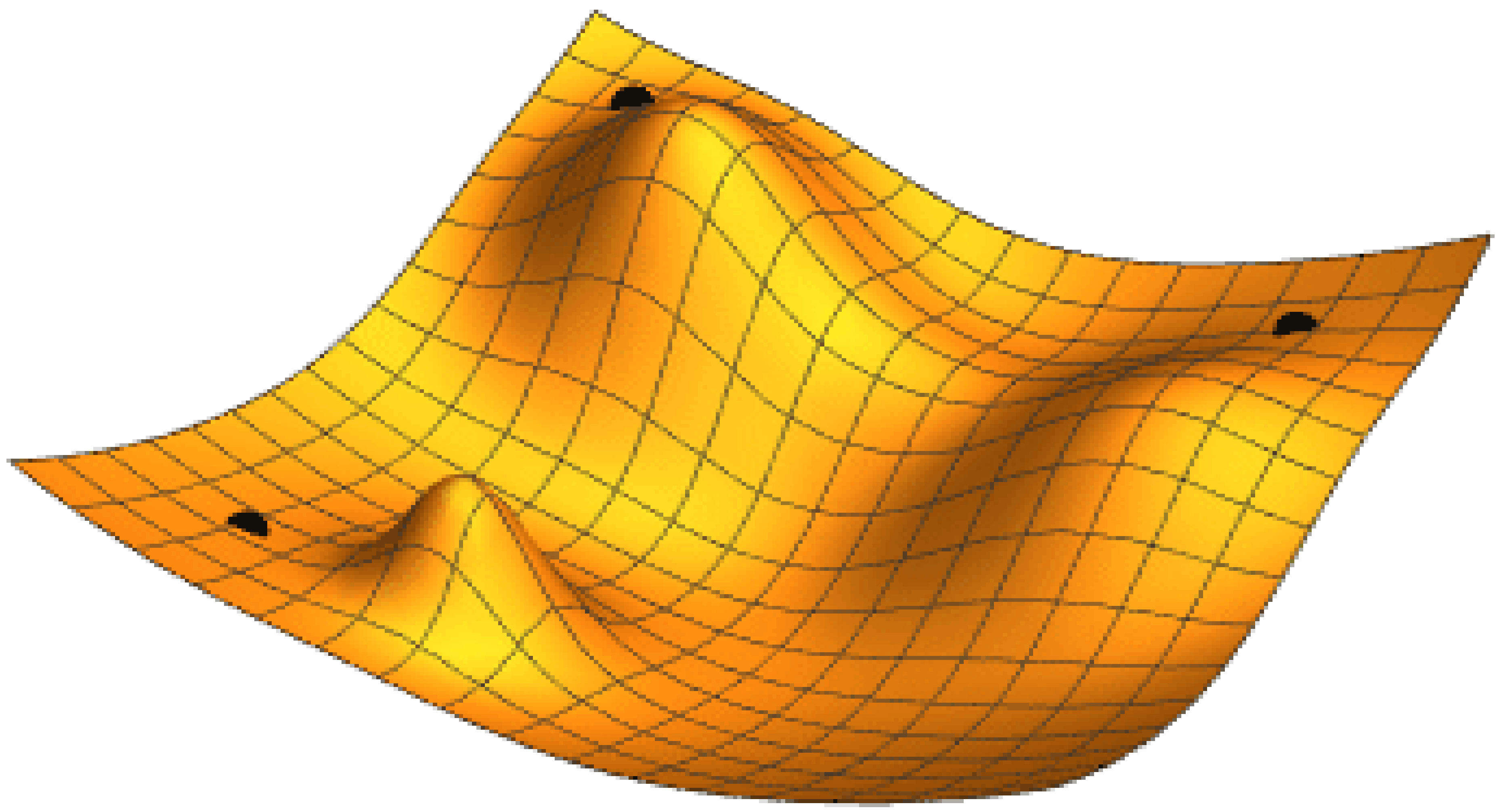
$\Rightarrow$ Löse Optimierungsproblem:

$$\mathrm{argmin}_{\omega \in \Omega}(\mathrm{loss} \circ \mathrm{neuralNet}(\omega; \mathrm{data}))$$

Gradient Descent

$$w_{n+1} = w_n - \alpha \cdot \frac{\partial f}{\partial w}$$





Abbildungen von: http://citadel.sjfc.edu/faculty/kgreen/vector/Block2/pder/node8.html , https://upload.wikimedia.org/wikipedia/commons/a/a3/Gradient_descent.gif

Automatic Differentiation

$$nn = l_L \circ \ldots \circ l_1$$

$$\Rightarrow \text{Ableitung: } Dnn = Dl_L \circ \ldots \circ Dl_1$$

- Funktionskomposition ist assoziativ, erlaubt Auswertung in beliebiger Reihenfolge
- Aufwand "vorwärts" abhängig von Eingangsdimension (hier: Anzahl der Parameter; heute bis 10^{11})
- Aufwand "rückwärts" abhängig von Ausgangsdimension (hier: 1)

Reverse Automatic Differentiation

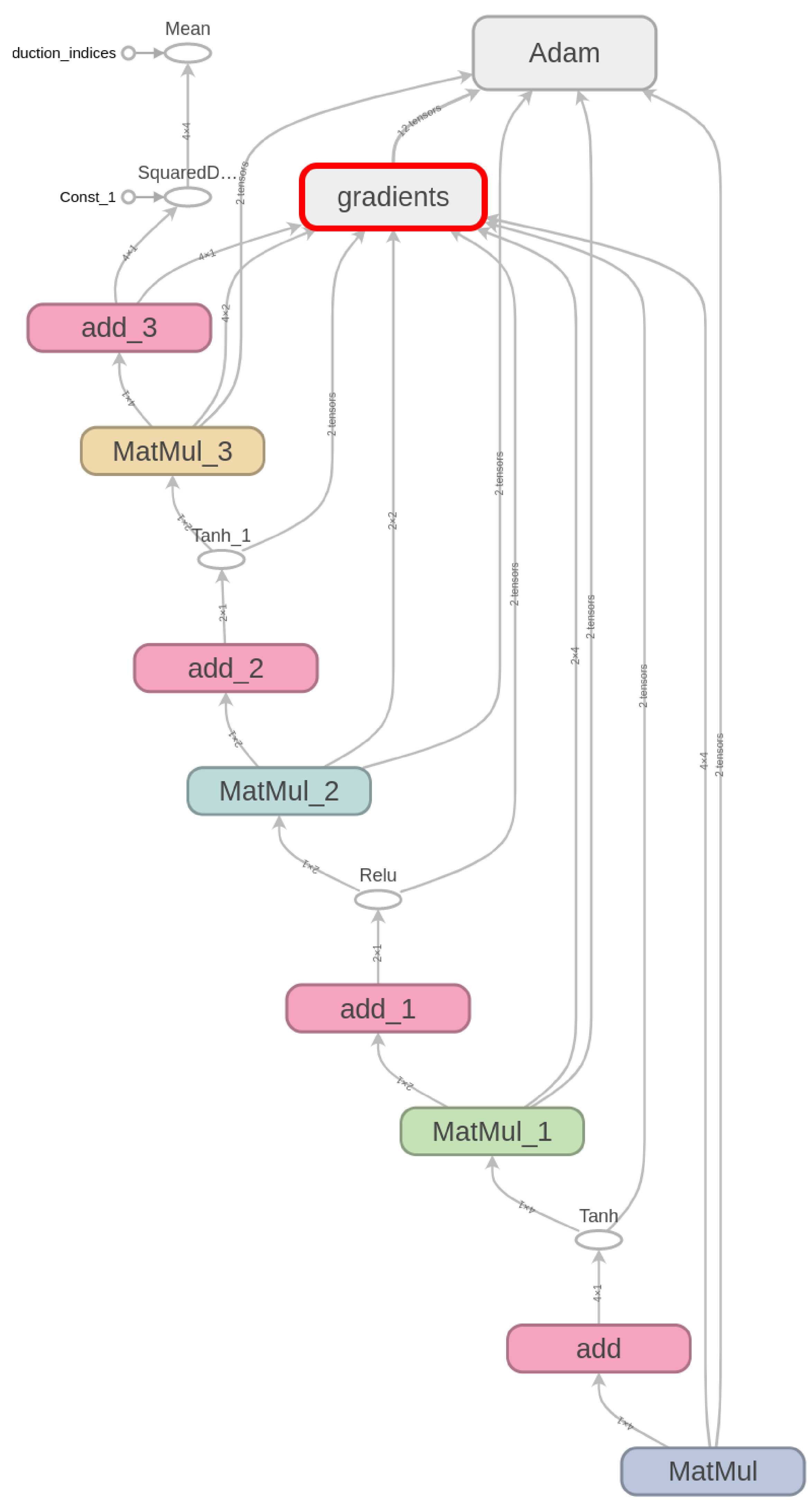
$$Dnn(v) = Dl_L(l_{L-1}(\dots)) \circ \dots \circ Dl_1(v)$$

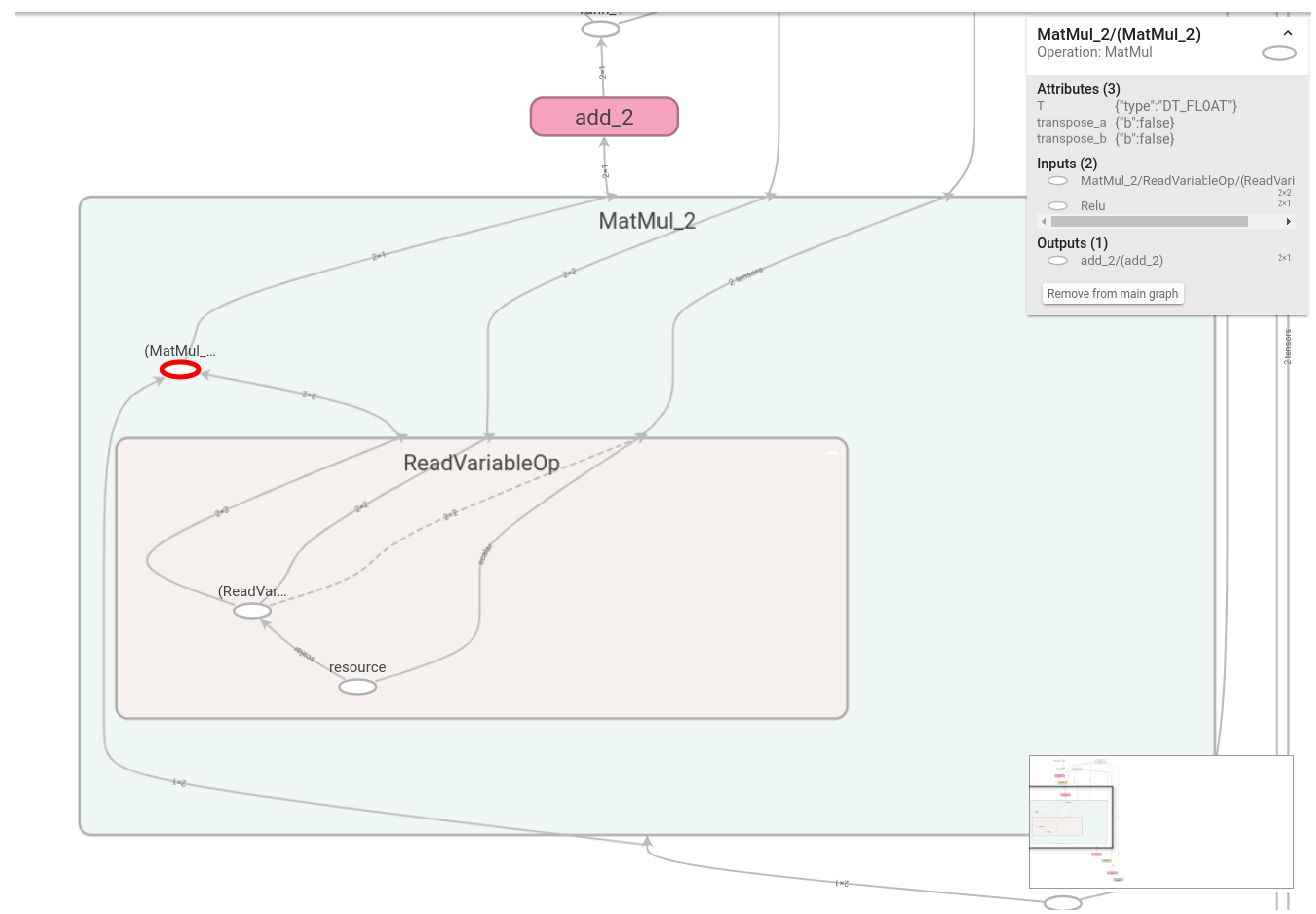
Um Dl_i zu berechnen, müssen wir die Ausgabe von l_{i-1} kennen

$\Rightarrow$ "Wengert-Liste"

Deep Learning Bibliotheken sind im Wesentlichen Werkzeuge zur Generierung und Verwaltung von Berechnungsgraphen.

TensorFlow Graphen





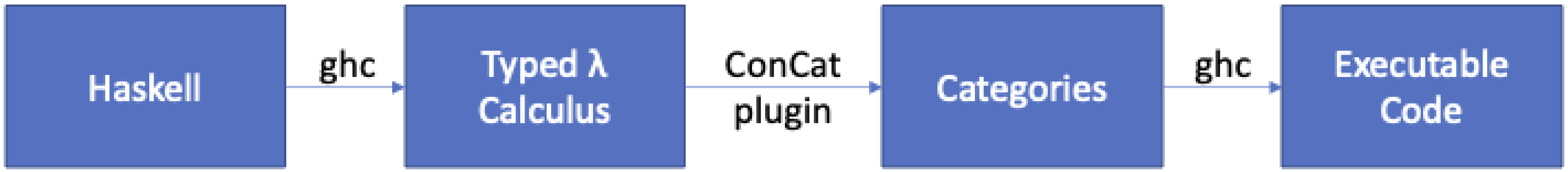
Implementierung in TensorFlow

```
class SimpleNN:

    def __init__(self, dimIn dimOut):
        self.dims = [dimIn, dimIn, dimIn, dimOut, dimOut]
        self.weights = []
        self.biases = []
        for i in range(4):
            self.weights.append(
                tf.Variable(tf.random.normal(shape=(self.dims[i+1],self.dims[i])))
                )
            self.biases.append(
                tf.Variable(tf.random.normal(shape=(self.dims[i+1],1)))
                )

    def __call__(self, x):
        inputs = tf.convert_to_tensor([x], dtype=tf.float32)
        out = tf.matmul(self.weights[0],
                        inputs, transpose_b=True) + self.biases[0]
        out = tf.tanh(out)
        out = tf.matmul(self.weights[1], out) + self.biases[1]
        out = tf.nn.relu(out)
        out = tf.matmul(self.weights[2], out) + self.biases[2]
        out = tf.tanh(out)

        return tf.matmul(self.weights[3], out) + self.biases[3]
```

Neuronale Netze mit ConCat

```
simpleNN ::
  ( KnownNat m,
    KnownNat n,
    Functor f,
    Foldable f,
    Floating num,
    ...
  ) =>
  SimpleNNParameters f m n num ->
  f m num ->
  f n num
simpleNN =
  affine
  @. affTanh
  @. affRelu
  @. affTanh
```

ConCat Funktionsweise



- Nutzt Isomorphie zwischen Lambda-Kalkülen und kartesisch abgeschlossenen Kategorien (CCC) [1]
- Übersetzt Haskell-Core in kategorielle Sprache
- Ausdrücke in kategorieller Sprache können in beliebigen CCCs interpretiert werden
- Abstrahiert dadurch Haskells Funktionspfeil `(->)`

[1] Joachim Lambek "Cartesian Closed Categories and Typed Lambda-calculi", In 13th Spring School on Combinators and Functional Programming Languages, 1985

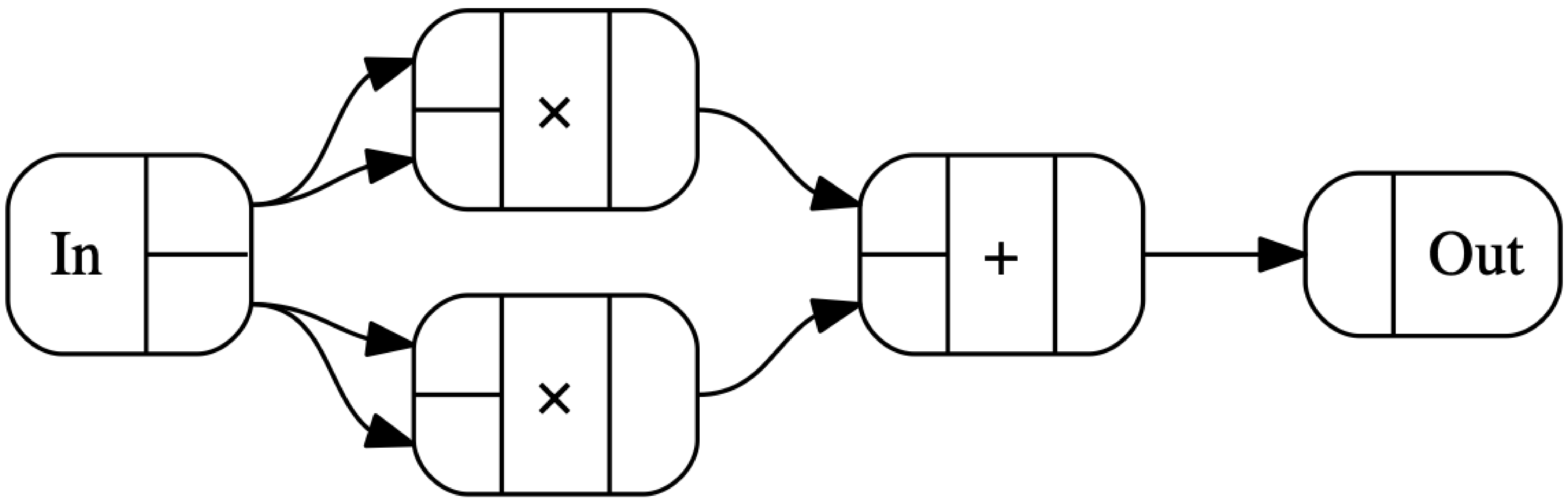
Beispiel einer ConCat-Transformation

```
magSqr :: Num a => (a, a) -> a
magSqr (a, b) = sqr a + sqr b
```

$\Rightarrow$ ConCat:

$$magSqr = addC \circ (mulC \circ (exl \triangle exl) \triangle mulC \circ (exr \triangle exr))$$

In Kategorie der Graphen – `(a, a) `Graph` a`:



Grafik von: Conal Elliot "The Simple Essence of Automatic Differentiation", ICFP 2018

@ctive group

Generalized Derivatives

Idee: Ergänze Funktion um ihre Ableitung

$$a \mapsto f(a) \Rightarrow a \mapsto (f(a), f'(a))$$

Kategorie der generalisierten Ableitungen:

```
newtype GD k a b = D {unD :: a -> b :* (a `k` b)}
```

Komposition für Generalized Derivatives

$$a \mapsto (f(a), f'(a))$$

```
instance Category k => Category (GD k) where
  ...
  D g . D f =
    D (\ a ->
            let (b, f') = f a
                (c, g') = g b
             in (c, g' . f')
        )
```

Kettenregel: $(g \circ f)'(x) = g'(f(x)) \circ f'(x)$

Multiplikation für Generalized Derivatives

$$a \mapsto (f(a), f'(a))$$

```
instance (LinearCat k s, Additive s, Num s) => NumCat (GD k) s where
  ...
  mulC    = D (mulC &&& \ (u,v) -> scale v |||| scale u)
```

$$\text{Produktregel: } (f \cdot g)'(x) = f'(x) \cdot g(x) + f(x) \cdot g'(x)$$

Forward Automatic Differentiation

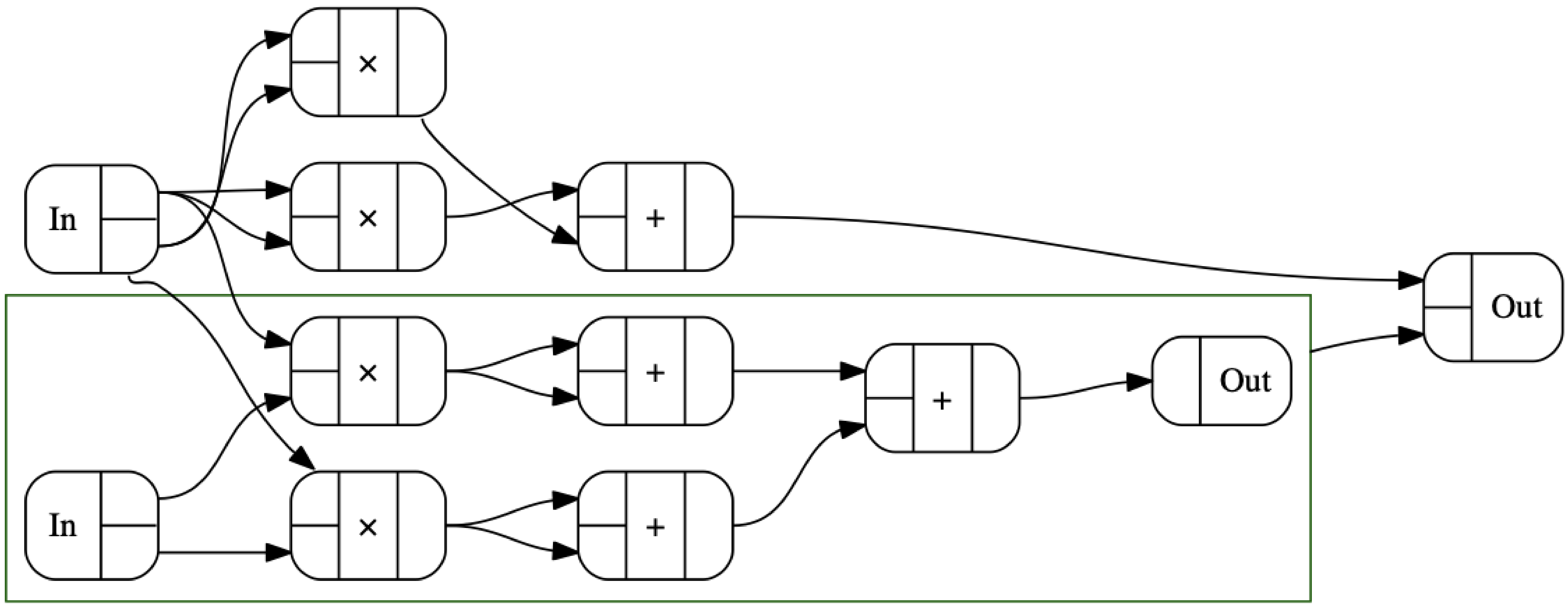


Figure 1: `magSqr in GD (-+>)`

Duale Kategorien

... entstehen durch Umdrehen aller Pfeile:

$$a \to b \Rightarrow b \to a$$

In Haskell:

```
newtype Dual k a b = Dual (b `k` a)
```

Beispiele Dualer Morphismen

```haskell
instance Category k => Category (Dual k) where
  ...
  -- flip :: (a -> b -> c) -> b -> a -> c
  (.) = inAbst2 (flip (.))

instance CoproductPCat k => ProductCat (Dual k) where
  ...
  -- exl :: (a, b) -> a; inlP :: a -> (a, b)
  exl = abst inlP
```

Reverse Automatic Differentiation

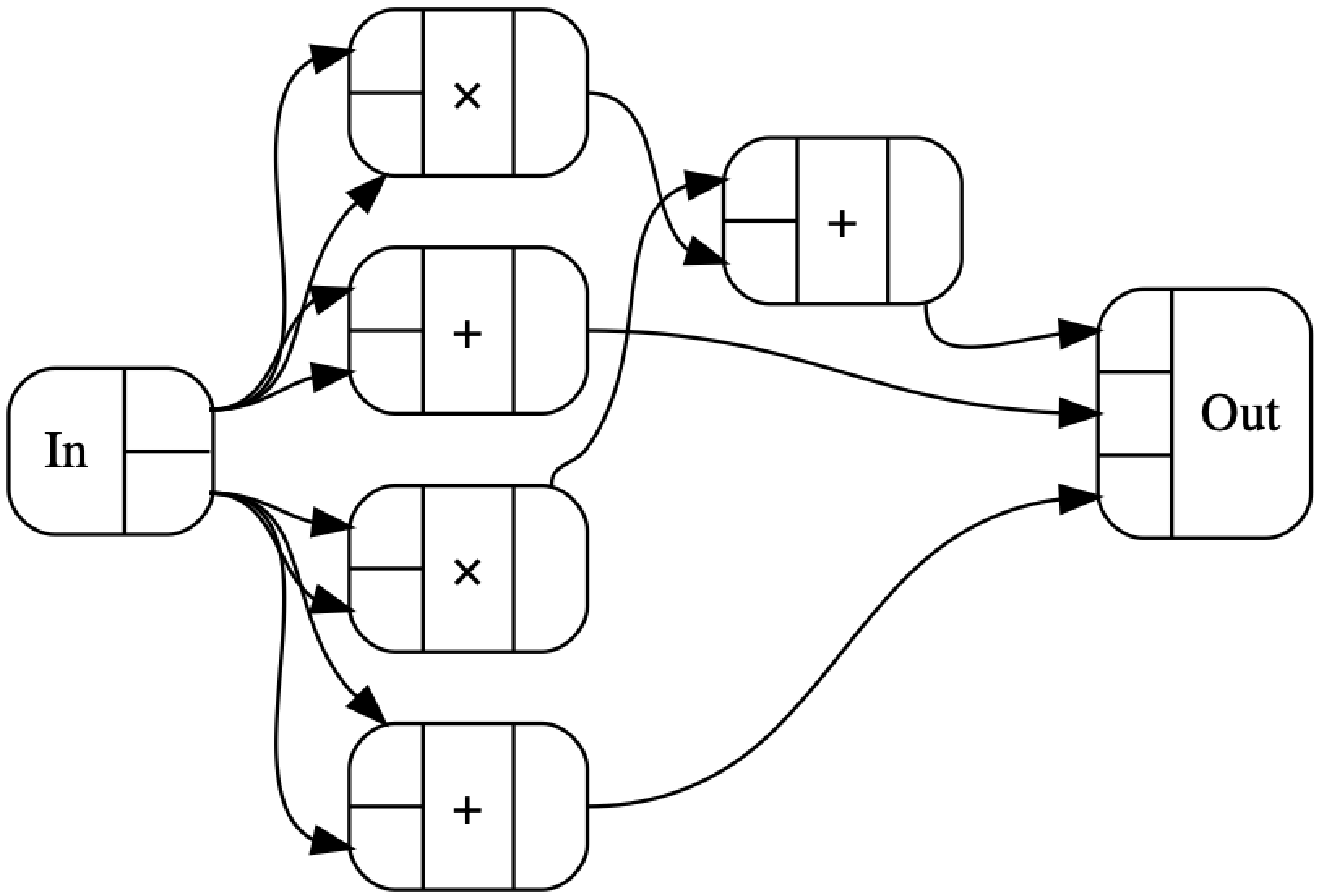


Figure 2: `magSqr in GD (Dual(-+>))`

Graphenfreie Gradienten

```
type RAD = GD (Dual (-+>))

grad :: Num s => (a -> s) -> (a -> a)
grad = friemelOutGrad . toCcc @RAD

nnGrad :: parameters -> parameters
nnGrad = grad (loss . nn)
```

Beschleunigtes Deep Learning in Haskell

> *"Data.Array.Accelerate defines an embedded array language for computations for high-performance computing in Haskell. [...] These computations may then be online compiled and executed on a range of architectures."*

Kategorie der Accelerate-Funktionen:

```
newtype AccFun a b where
  AccFun :: (AccValue a -> AccValue b) -> AccFun a b
```

ConCelerate: ConCat + Accelerate

```
simpleNN :: (SimpleNNConstraints f m n num) => SimpleNN f m n num
simpleNN = affine @. affTanh @. affRelu @. affTanh

simpleNNGrad ::
   (KnownNat m, KnownNat n) =>
   (Vector m Double, Vector n Double) ->
  SimpleNNParameters m n Double ->
  SimpleNNParameters m n Double
simpleNNGrad = errGrad simpleNN

simpleNNGradAccFun ::
   (KnownNat m, KnownNat n) =>
   (Vector m Double, Vector n Double) ->
  SimpleNNParameters m n Double `AccFun` SimpleNNParameters m n Dou
simpleNNGradAccFun labeledInput = toCcc (simpleNNGrad labeledInput)
```

Vielen Dank!

ConCat

Accelerate

Active Group

@ctive group