Neural Nets with Automatic Differentiation

https://github.com/brunjlar/neural

About Me

Lars Brünjes (PhD)

(Pure) Mathematician, Lead Software Architect

based in Regensburg, Germany

Email: brunjlar@gmail.com

Github: https://github.com/brunjlar

Agenda

- Artificial Neural Networks
- Gradient Descent
- Backpropagation
- The ***neurals*** library
    - Automatic Differentiation
    - Components
    - Layers
    - Pipes
    - Other Features
    - Examples
- Questions & Comments

Agenda

- Artificial Neural Networks
- Gradient Descent
- Backpropagation
- The ***neurals*** library
  - Automatic Differentiation
  - Components
  - Layers
  - Pipes
  - Other Features
  - Examples
- Questions & Comments

on Hackage...

neural: Neural Networks in native Haskell | Home | Search

The neural package

[Tags:benchmark, library, mit, program, test]

The goal of `neural` is to provide a modular and flexible neural network library written in native Haskell.

Features include

- *composability* via arrow-like instances and pipes,
- *automatic differentiation* for automatic gradient descent/ backpropagation training (using Edward Kmett's fabulous ad library).

The idea is to be able to easily define new components and wire them up in flexible, possibly complicated ways (convolutional deep networks etc.).

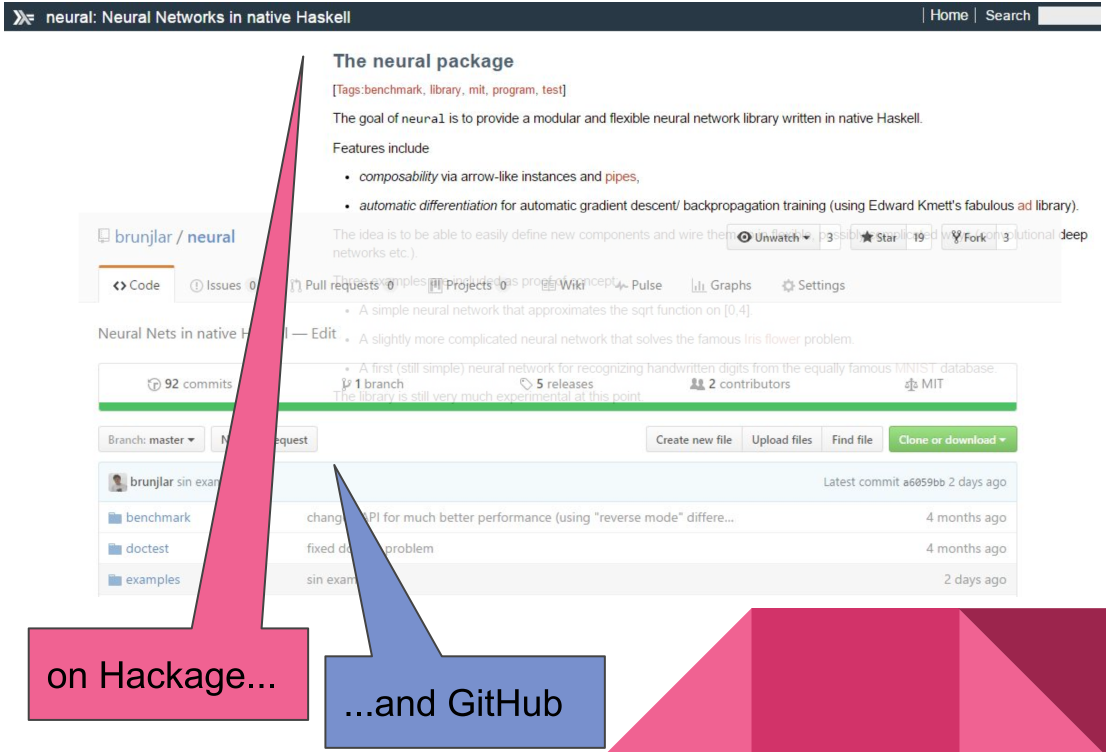
Three examples are included as proof of concept:

- A simple neural network that approximates the sqrt function on [0,4].
- A slightly more complicated neural network that solves the famous Iris flower problem.
- A first (still simple) neural network for recognizing handwritten digits from the equally famous MNIST database.

The library is still very much experimental at this point.

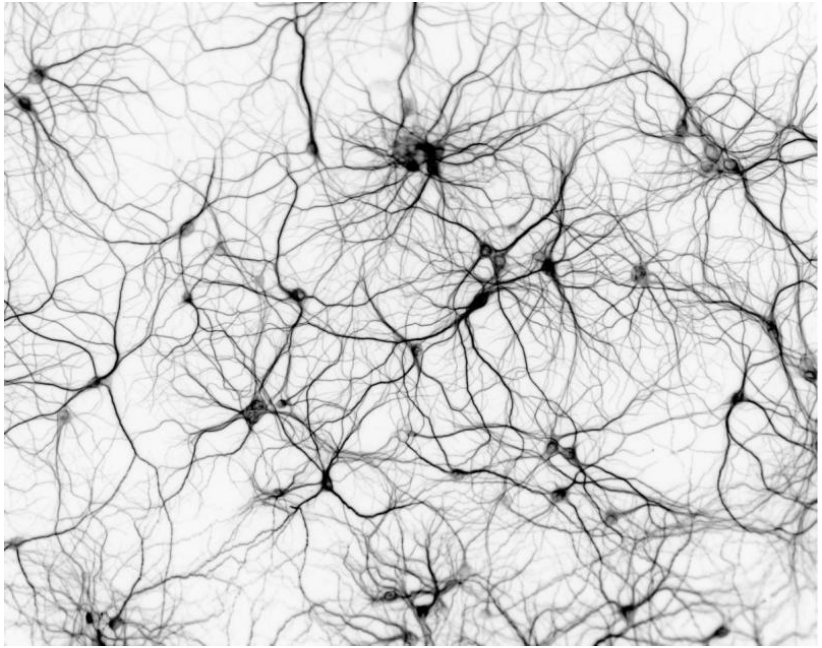
Agenda

- Artificial Neural Networks
- Gradient Descent
- Backpropagation
- The ***neurals*** library
  - Automatic Differentiation
  - Components
  - Layers
  - Pipes
  - Other Features
  - Examples
- Questions & Comments



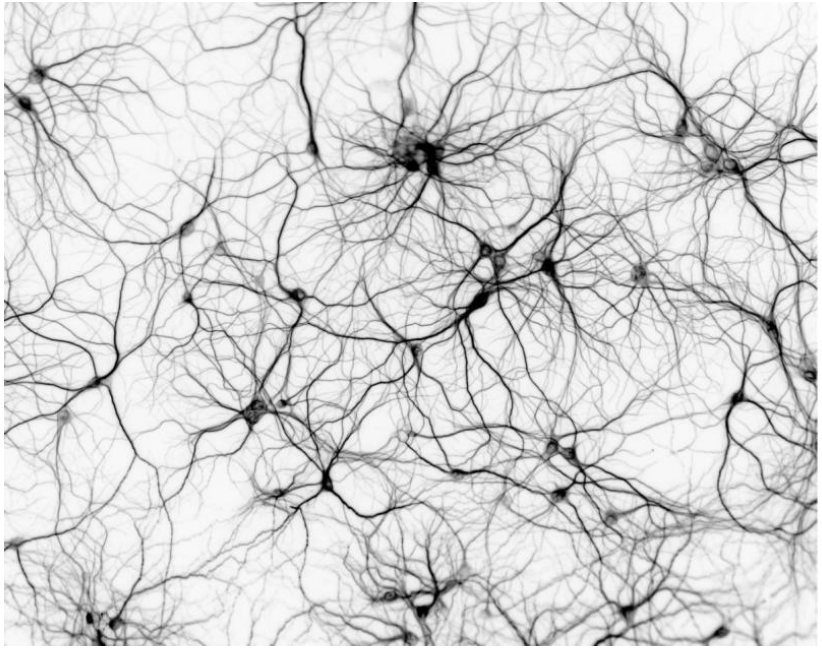
Artificial Neural Networks

Artificial Neural Networks



Taking inspiration from biological brains to approach Artificial Intelligence.

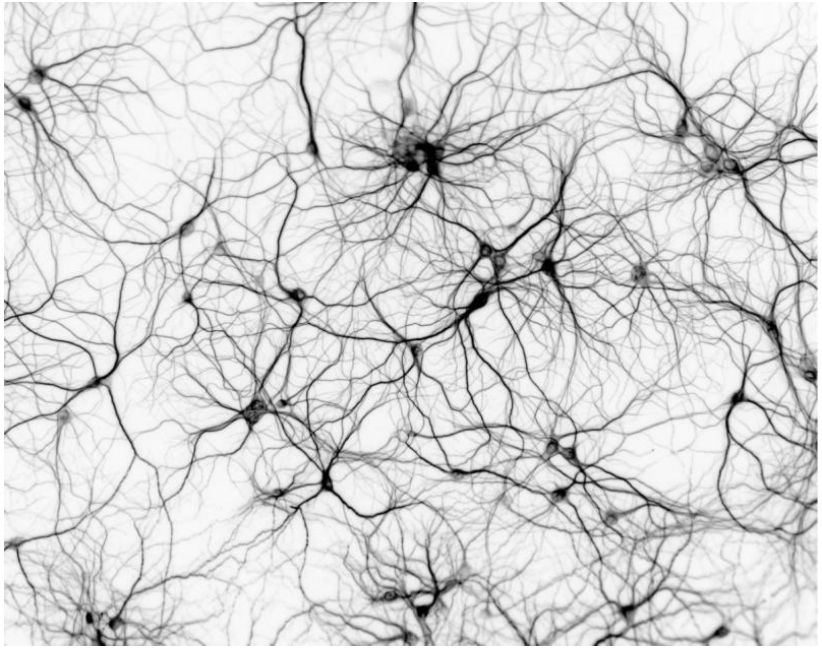
Artificial Neural Networks



Taking inspiration from biological brains to approach Artificial Intelligence.

- 1943 McCulloch-Pitts Neuron

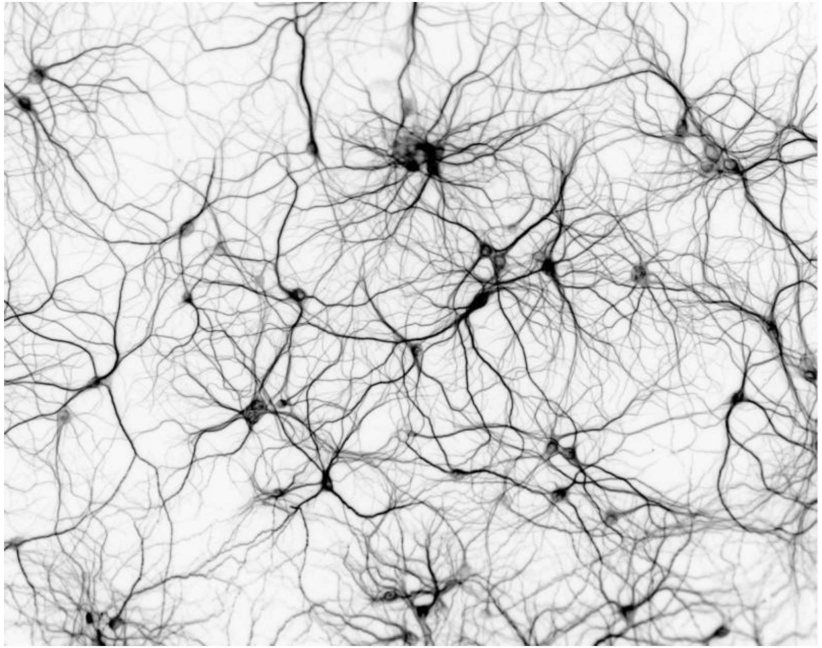
Artificial Neural Networks



Taking inspiration from biological brains to approach Artificial Intelligence.

- 1943 McCulloch-Pitts Neuron
- 1958 Perceptron

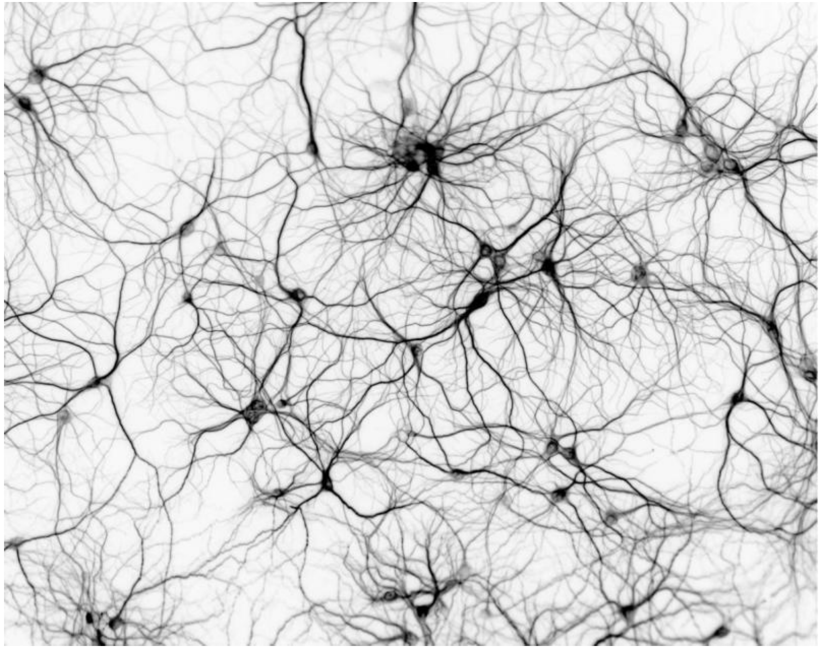
Artificial Neural Networks



Taking inspiration from biological brains to approach Artificial Intelligence.

- 1943 McCulloch-Pitts Neuron
- 1958 Perceptron
- 1975 Backpropagation

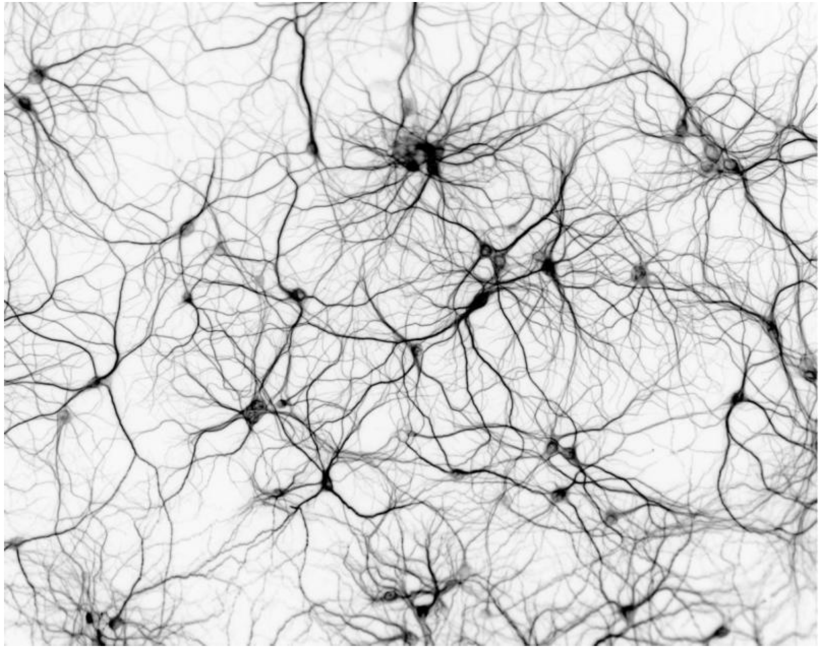
Artificial Neural Networks



Taking inspiration from biological brains to approach Artificial Intelligence.

- 1943 McCulloch-Pitts Neuron
- 1958 Perceptron
- 1975 Backpropagation
- 2006 Deep Learning

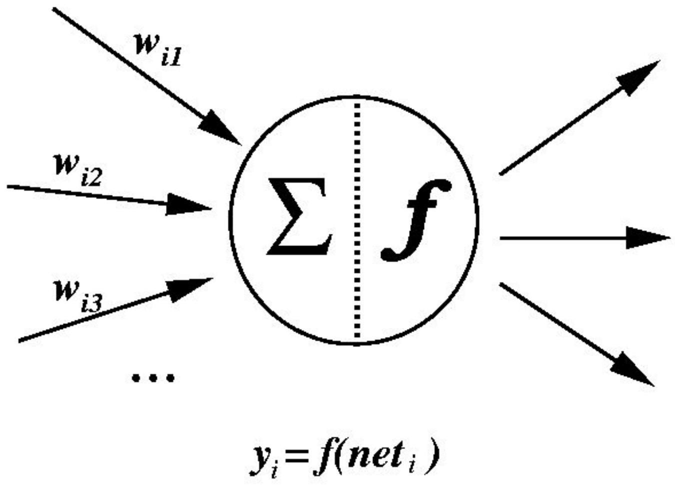
Artificial Neural Networks



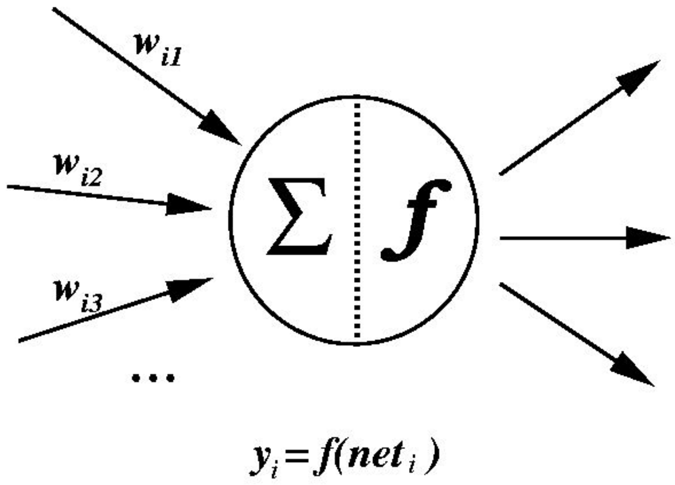
Taking inspiration from biological brains to approach Artificial Intelligence.

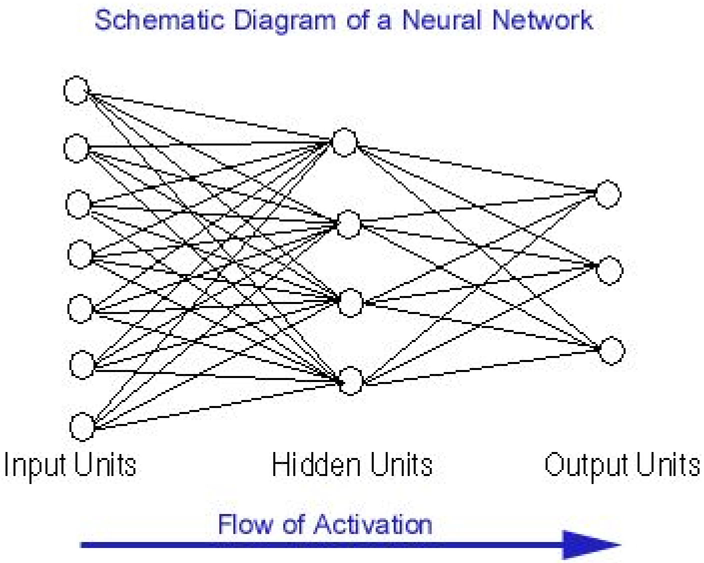
- 1943 McCulloch-Pitts Neuron
- 1958 Perceptron
- 1975 Backpropagation
- 2006 Deep Learning
- 2015 AlphaGo

Artificial Neural Networks

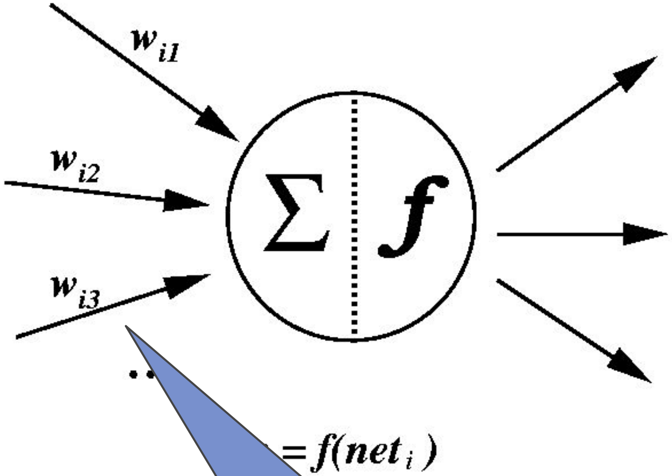


Artificial Neural Networks

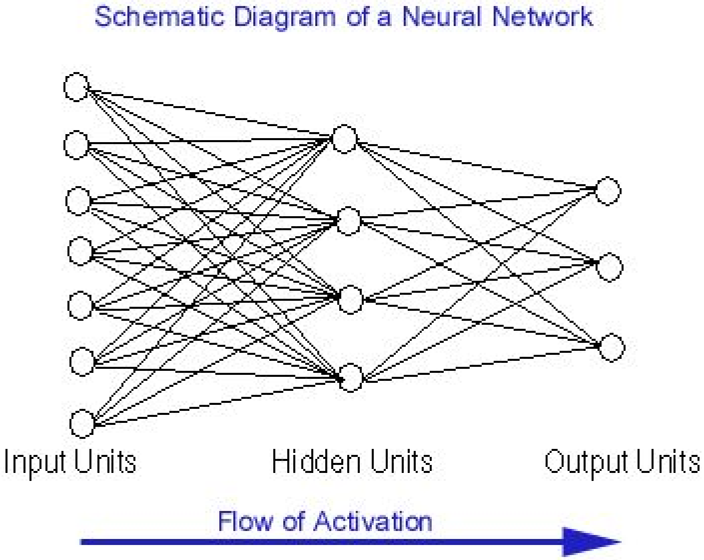




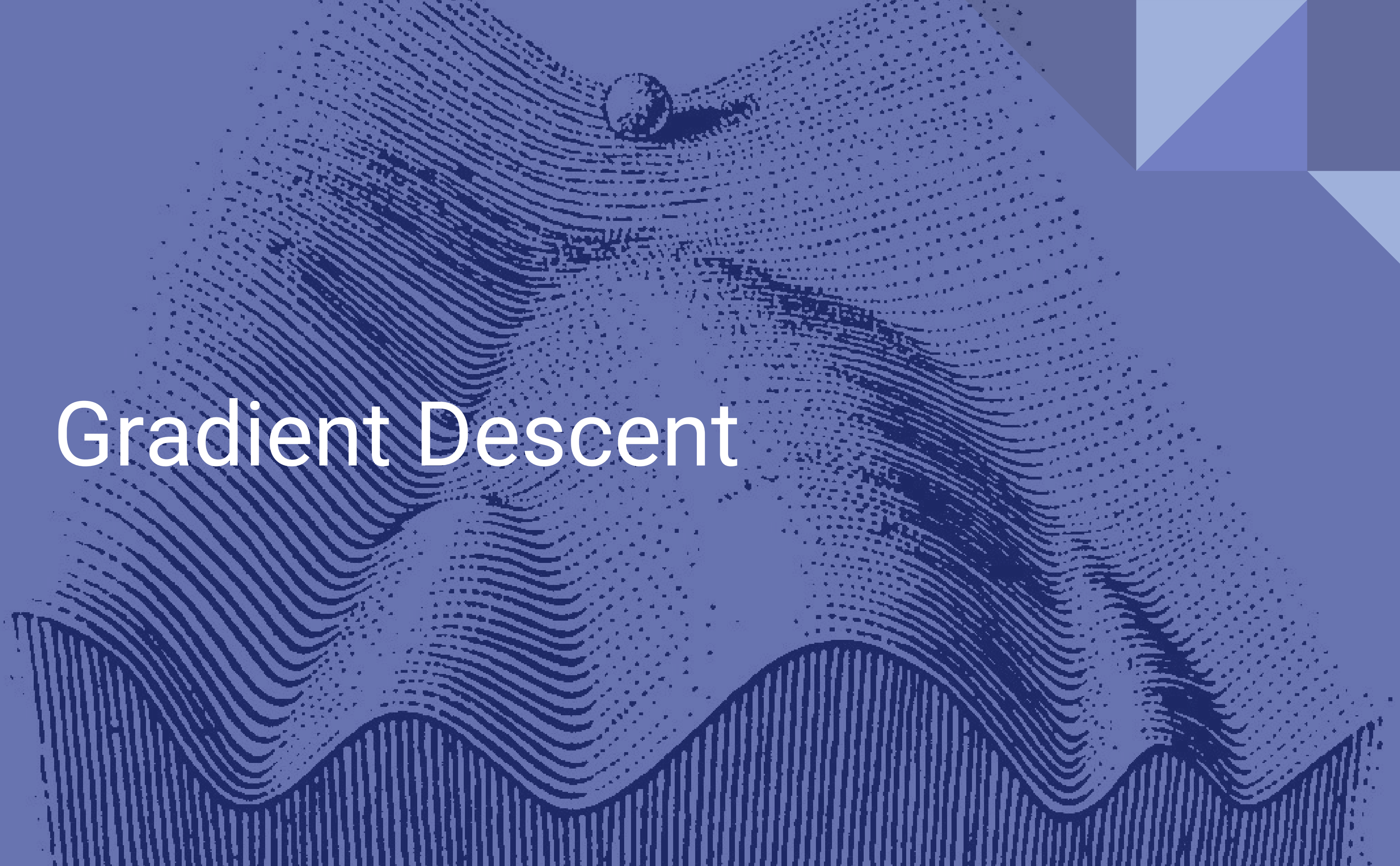
Artificial Neural Networks



The network learns by adjusting the *weights*



Gradient Descent



Gradient Descent

Extremely generic method for finding minima (or maxima) of differentiable functions:

Gradient Descent

Extremely generic method for finding minima (or maxima) of differentiable functions:

We want to **minimize** the **error** of our artificial neural networks!

Gradient Descent

Extremely generic method for finding minima (or maxima) of differentiable functions:

- The “gradient” points into the direction of steepest incline. $\nabla f = \left(\frac{\partial f}{\partial x_i} \right)_i$

Gradient Descent

Extremely generic method for finding minima (or maxima) of differentiable functions:

- The “gradient” points into the direction of steepest incline. $\nabla f = \left(\frac{\partial f}{\partial x_i} \right)_i$

- At each step, move into the opposite direction.

Gradient Descent

Extremely generic method for finding minima (or maxima) of differentiable functions:

- The “gradient” points into the direction of steepest incline. $\nabla f = \left(\frac{\partial f}{\partial x_i} \right)_i$
- At each step, move into the opposite direction.
- Eventually, you’ll end up at a (local) minimum.

Gradient Descent

Extremely generic method for finding minima (or maxima) of differentiable functions:

- The "gradient" points into the direction of steepest incline. $\nabla f = \left(\frac{\partial f}{\partial x_i} \right)_i$
- At each step, move into the opposite direction.
- Eventually, you'll end up at a (local) minimum.

All we need is to compute the gradient!

Backpropagation

Backpropagation...

$$\delta_{Mi} = -\frac{\partial E_p}{\partial O_{pi}} \frac{\partial Y_{Mi}}{\partial net_{Mi}} \quad (2.5)$$

$$= -\frac{\partial}{\partial O_{pi}}\left[\frac{1}{2}(T_{p1}-O_{p1})^2 + \dots + (T_{pi}-O_{pi})^2 + \dots\right] \frac{\partial}{\partial net_{Mi}} f(net_{Mi}) \quad (2.6)$$

$$= -\frac{\partial}{\partial O_{pi}}\left[\frac{1}{2}(T_{pi}-O_{pi})^2\right] f'(net_{Mi})$$

$$\delta_{Mi} = (T_{pi}-O_{pi})[f(net_{Mi})[1-f(net_{Mi})]] \quad (2.8)$$

$$= (T_{pi}-O_{pi})(O_{pi})(1-O_{pi}) \quad (2.9)$$

$$\frac{\partial E_p}{\partial Y_{ji}} = \frac{\partial E_p}{\partial net_{(j+1)1}} \frac{\partial net_{(j+1)1}}{\partial Y_{ji}} + \frac{\partial E_p}{\partial net_{(j+1)2}} \frac{\partial net_{(j+1)2}}{\partial Y_{ji}} + \dots \quad (2.10)$$

$$= \sum_{a=1}^{N_{j+1}} \left[\frac{\partial E_p}{\partial net_{(j+1)a}} \frac{\partial net_{(j+1)a}}{\partial Y_{ji}}\right] \quad (2.11)$$

$$= \sum_{a=1}^{N_{j+1}} \left[-\delta_{(j+1)a} \frac{\partial}{\partial Y_{ji}}(W_{(j+1)a0}Y_{j0} + \dots + W_{(j+1)ai}Y_{ji} + \dots)\right] \quad (2.12)$$

$$= \sum_{a=1}^{N_{j+1}} \left[-\delta_{(j+1)a} \frac{\partial}{\partial Y_{ji}}(W_{(j+1)ai}Y_{ji})\right] \quad (2.13)$$

$$= \sum_{a=1}^{N_{j+1}} \left[-\delta_{(j+1)a} W_{(j+1)ai}\right]. \quad (2.14)$$

$$\delta_{ji} = -\sum_{a=1}^{N_{j+1}} \left[-\delta_{(j+1)a} W_{(j+1)ai}\right] \frac{\partial Y_{ji}}{\partial net_{ji}}$$

$$= Y_{ji}(1-Y_{ji}) \sum_{a=1}^{N_{j+1}} \left[\delta_{(j+1)a} W_{(j+1)ai}\right] \quad (2.16)$$

...is just Gradient Descent

Mathematically simple, a rather tedious application of the Chain Rule.

Backpropagation

But details change when...

Backpropagation

But details change when...

- A different activation function is used for neurons.

Backpropagation

But details change when...

- A different activation function is used for neurons.
- A different error function is used.

Backpropagation

But details change when...

- A different activation function is used for neurons.
- A different error function is used.
- A different layout is used.

Backpropagation

Furthermore, the classical formulas are **index-based**, i.e. complicated when network topologies become complex.

Backpropagation

Furthermore, the classical formulas are **index-based**, i.e. complicated when network topologies become complex.

$$\nabla f = \left(\frac{\partial f}{\partial x_i} \right)_i$$

Automatic Differentiation

Automatic Differentiation

There are two well-known ways to use a computer for differentiation:

Automatic Differentiation

There are two well-known ways to use a computer for differentiation:

- **Numeric Approximation**, basically $f'(x) \approx \frac{f(x+h)-f(x)}{h}$

Automatic Differentiation

There are two well-known ways to use a computer for differentiation:

- **Numeric Approximation**, basically $f'(x) \approx \frac{f(x+h)-f(x)}{h}$
- **Symbolic Differentiation**, manipulating abstract algebraic expressions.

Automatic Differentiation

There are two well-known ways to use a computer for differentiation:

- **Numeric Approximation**, basically $f'(x) \approx \frac{f(x+h)-f(x)}{h}$
- **Symbolic Differentiation**, manipulating abstract algebraic expressions.

Less well known is the third way, **Automatic Differentiation** (see Conal M. Elliott's fantastic paper "Beautiful Differentiation"), an **exact numeric** method.

Automatic Differentiation

There are two well-known ways to use a computer for differentiation:

- **Numeric Approximation**, basically $f'(x) \approx \frac{f(x+h)-f(x)}{h}$
- **Symbolic Differentiation**, manipulating abstract algebraic expressions.

Less well known is the third way, **Automatic Differentiation** (see Conal M. Elliott’s fantastic paper “Beautiful Differentiation”), an **exact numeric** method.

Basic Idea: Manipulate numbers and their derivatives simultaneously (can be done conveniently in Haskell by overloading the numeric type classes).

Edward Kmett's *ad* Library

```
grad :: (Traversable f, Num a) => (forall s. Reifies s Tape => f (Reverse s a) -> Reverse s a) -> f a -> f a
```
Source

The grad function calculates the gradient of a non-scalar-to-scalar function with reverse-mode AD in a single pass.

```
>>> grad (\[x,y,z] -> x*y+z) [1,2,3]
[2,1,1]
```

```
grad' :: (Traversable f, Num a) => (forall s. Reifies s Tape => f (Reverse s a) -> Reverse s a) -> f a -> (a, f a)
```
Source

The grad' function calculates the result and gradient of a non-scalar-to-scalar function with reverse-mode AD in a single pass.

```
>>> grad' (\[x,y,z] -> x*y+z) [1,2,3]
(5,[2,1,1])
```

```
gradWith :: (Traversable f, Num a) => (a -> a -> b) -> (forall s. Reifies s Tape => f (Reverse s a) -> Reverse s a) -> f a -> f b
```
Source

grad g f function calculates the gradient of a non-scalar-to-scalar function f with reverse-mode AD in a single pass. The gradient is combined element-wise with the argument using the function g.

```
grad == gradWith (_ dx -> dx)
id == gradWith const
```

```
gradWith' :: (Traversable f, Num a) => (a -> a -> b) -> (forall s. Reifies s Tape => f (Reverse s a) -> Reverse s a) -> f a -> (a, f b)
```
Source

grad' g f calculates the result and gradient of a non-scalar-to-scalar function f with reverse-mode AD in a single pass the gradient is combined element-wise with the argument using the function g.

```
grad' == gradWith' (_ dx -> dx)
```

Edward Kmett's *ad* Library

```
grad :: (Traversable f, Num a) => (forall s. Reifies s Tape => f (Reverse s a) -> Reverse s a) -> f a -> f a
```
\# Source

The grad function calculates the gradien[illegible]alar-to-scalar function with reverse-mode AD in a single pass.

```
>>> grad (\[x,y,z] -> x*y+z) [1,2,3]
[2,1,1]
```

```
grad' :: (Traversable f, Num a) => (forall s. Reifies s Tape => f (Reverse s a) -> Reverse s a) -> f a -> (a, f a)
```
\# Source

The grad' function calculates the result and gradient of a non-scalar-to-scalar function with reverse-mode AD in a single pass.

```
>>> grad' (\[x,y,z] -> x*y+z) [1,2,3]
(5,[2,1,1])
```

```
gradWith :: (Traversable f, Num a) => (a -> a -> b) -> (forall s. Reifies s Tape => f (Reverse s a) -> Reverse s a) -> f a -> f b
```
\# Source

grad g f function calculates the gradient of a non-scalar-to-scalar function f with reverse-mode AD in a single pass. The gradient is combined element-wise with the argument using the function g.

```
grad == gradWith (_ dx -> dx)
id == gradWith const
```

```
gradWith' :: (Traversable f, Num a) => (a -> a -> b) -> (forall s. Reifies s Tape => f (Reverse s a) -> Reverse s a) -> f a -> (a, f b)
```
\# Source

grad' g f calculates the result and gradient of a non-scalar-to-scalar function f with reverse-mode AD in a single pass the gradient is combined element-wise with the argument using the function g.

```
grad' == gradWith' (_ dx -> dx)
```

Arguments of "traversable" shape - no indices needed!

Edward Kmett's *ad* Library

```
grad :: (Traversable f, Num a) => (forall s. Reifies s Tape => f (Reverse s a) -> Reverse s a) -> f a -> f a
```
Source

The grad function calculates the gradient ... scalar-to-scalar function with reverse-mode AD in a single pass.

```
>>> grad (\[x,y,z] -> x*y+z) [1,2,3]
[2,1,1]
```

```
grad' :: (Traversable f, Num a) => (forall s. Reifies s Tape => f (Reverse s a) -> Reverse s a) -> f a -> (a, f a)
```
Source

The grad' function calculates the result and gradient of a non-scalar-to-scalar function with reverse-mode AD in a single pass.

```
>>> grad' (\[x,y,z] -> x*y+z) [1,2,3]
(5,[2,1,1])
```

```
gradWith :: (Traversable f, Num a) => (a -> a -> b) -> (forall s. Reifies s Tape => f (Reverse s a) -> Reverse s a) -> f a -> f b
```
Source

grad g f function calculates the gradient of a non-scalar-to-scalar function f with reverse-mode AD in a single pass. The gradient is combined element-wise with the argument using the function g.

```
grad == gradWith (_ dx -> dx)
id == gradWith const
```

```
gradWith' :: (Traversable f, Num a) => (a -> a -> b) -> (forall s. Reifies s Tape => f (Reverse s a) -> Reverse s a) -> f a -> (a, f b)
```
Source

grad' g f calculates the result and gradient of a non-scalar-to-scalar function f with reverse-mode AD in a single pass the gradient is combined element-wise with the argument using the function g.

```
grad' == gradWith' (_ dx -> dx)
```

Arguments of "traversable" shape - no indices needed!

These types look intimidating...

Neural Wrapper around *ad*

```haskell
class (Floating a, Ord a) => Analytic a where
```

Class `Analytic` is a helper class for defining differentiable functions.

Minimal complete definition

```haskell
fromDouble
```

Methods

```haskell
fromDouble :: Double -> a
```

Instances

`Analytic Double`	# Sou
`Reifies * s Tape => Analytic (Reverse s Double)`	# Sou

Neural Wrapper around *ad*

```
class (Floating a, Ord a) => Analytic a where
```

Class `Analytic` is a helper class for defining differentiable functions.

Minimal complete definition

```
fromDouble
```

Methods

```
fromDouble :: Double -> a
```

Instances

```
Analytic Double                                        # Sou
Reifies * s Tape => Analytic (Reverse s Double)        # Sou
```

There are those scary types again!

Neural Wrapper around *ad*

```
newtype Diff f g
```

Type `Diff f g` can be thought of as the type of "differentiable" functions `f Double -> g Double`.

Constructors

```
Diff
    runDiff :: forall a. Analytic a => f a -> g a
```

Instances

```
Category (* -> *) Diff    # Source
```

```
type Diff' = forall a. Analytic a => a -> a
```

Type `Diff'` can be thought of as the type of differentiable functions `Double -> Double`.

Neural Wrapper around *ad*

```
gradWith'
```

`:: Traversable t`	
`=> (Double -> Double -> a)`	how to combine argument and gradient
`-> Diff t Identity`	differentiable function
`-> t Double`	function argument
`-> (Double, t a)`	function value and combination of argument and gradient

Computes the gradient of an analytic function and combines it with the argument.

```
>>> gradWith' (\_ d -> d) (Diff $ \[x, y] -> Identity $ x * x + 3 * y + 7) [2, 1]
(14.0,[4.0,3.0])
```

Components

Components

```
newtype ParamFun s t a b                                # Sou
```

The type `ParamFun s t a b` describes parameterized functions from a to b, where the parameters are of type `t s`. When such components are composed, they all share the *same* parameters.

Constructors

```
ParamFun
    runPF :: a -> t s -> b
```

Instances

```
Category * (ParamFun s t)          # Source
Arrow (ParamFun s t)               # Source
ArrowChoice (ParamFun s t)         # Source
Profunctor (ParamFun s t)          # Source
ArrowConvolve (ParamFun s t)       # Source
Functor (ParamFun s t a)           # Source
Applicative (ParamFun s t a)       # Source
```

Components

```
newtype ParamFun s t a b                                # Sou
```

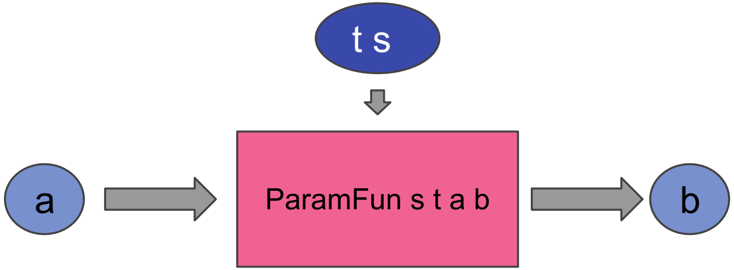
The type `ParamFun s t a b` describes parameterized functions from a to b, where the parameters are of type `t s`. When such components are composed, they all share the *same* parameters.

Constructors

```
ParamFun
    runPF :: a -> t s -> b
```

Instances

```
Category * (ParamFun s t)        # Source
Arrow (ParamFun s t)             # Source
ArrowChoice (ParamFun s t)       # Source
Profunctor (ParamFun s t)        # Source
ArrowConvolve (ParamFun s t)     # Source
Functor (ParamFun s t a)         # Source
Applicative (ParamFun s t a)     # Source
```



Components

```
data Component f g                                                    # Sourc
```

A `Component f g` is a parameterized differentiable function `f Double -> g Double`. In contrast to `ParamFun`, when components are composed, parameters are not shared. Each component carries its own collection of parameters instead.

Constructors

```
(Traversable t, Applicative t, NFData (t Double)) => Component
```

`weights :: t Double`	the specific parameter values
`compute :: forall s. Analytic s => ParamFun s t (f s) (g s)`	the encapsulated parameterized function
`initR :: forall m. MonadRandom m => m (t Double)`	randomly sets the parameters

Instances

```
NFData (Component f g)            # Source
Category (* -> *) Component       # Source
```

Components

`data Component f g` # Source

A `Component f g` is a parameterized differentiable function `f Double -> g Double`. In contrast to `ParamFun`, when components are composed, parameters are not shared. Each component carries its own collection of parameters instead.

Constructors

`(Traversable t, Applicative t, NFData (t Double)) => Component`

`weights :: t Double`	the specific parameter values
`compute :: forall s. Analytic s => ParamFun s t (f s) (g s)`	the encapsulated parameterized function
`initR :: forall m. MonadRandom m => m (t Double)`	randomly sets the parameters

Instances

`NFData (Component f g)`	# Source
`Category (* -> *) Component`	# Source

The traversable type t is “hidden” by existential quantification

Components

```
data Component f g                                                    # Source
```

A `Component f g` is a parameterized differentiable function `f Double -> g Double`. In contrast to `ParamFun`, when components are composed, parameters are not shared. Each component carries its own collection of parameters instead.

Constructors

```
(Traversable t, Applicative t, NFData (t Double)) => Component
```

`weights :: t Double`	the specific parameter values
`compute :: forall s. Analytic s => ParamFun s t (f s) (g s)`	the encapsulated parameterized function
`initR :: forall m. MonadRandom m => m (t Double)`	randomly sets the parameters

Instances

`NFData (Component f g)`	# Source
`Category (* -> *) Component`	# Source

The traversable type t is “hidden” by existential quantification

Traversables are composed automatically!

Components

```
data Model :: (* -> *) -> (* -> *) -> * -> * -> * -> * where
```
Source

A `Model f g a b c` wraps a `Component f g` and models functions `b -> c` with "samples" (for model error determination) of type a.

Constructors

```
Model :: (Functor f, Functor g) => Component f g -> (a -> (f Double, Diff g Identity)) -> (b -> f Double) -> (g Double -> c) -> Model f g a b c
```

Instances

- `Profunctor (Model f g a)` # Source
- `NFData (Model f g a b c)` # Source

Components

```
data Model :: (* -> *) -> (* -> *) -> * -> * -> * -> * where
```
Source

A `Model f g a b c` wraps a `Component f g` and models functions `b -> c` with "samples" (for model error determination) of type a.

Constructors

```
Model :: (Functor f, Functor g) => Component f g -> (a -> (f Double, Diff g Identity)) -> (b -> f Double) -> (g Double -> c) -> Model f g a b c
```

Instances

```
Profunctor (Model f g a)
```
Source

```
NFData (Model f g a b c)
```
Source

```
type StdModel f g b c = Model f g (b, c) b c
```

A type abbreviation for the most common type of models, where samples are just input-output tuples.

```
mkStdModel :: (Functor f, Functor g) => Component f g -> (c -> Diff g Identity) -> (b -> f Double) -> (g Double -> c) -> StdModel f g b c
```

Creates a `StdModel`, using the simplifying assumtion that the error can be computed from the expected output allone.

Components

descent

`:: Foldable h`	
`=> Model f g a b c`	the model whose error should be decreased
`-> Double`	the learning rate
`-> h a`	a mini-batch of samples
`-> (Double, Model f g a b c)`	returns the average sample error and the improved model

Performs one step of gradient descent/ backpropagation on the model,

Layers

Layers

```
type Layer i o = Component (Vector i) (Vector o)
```

A `Layer i o` is a component that maps a `Vector` of length `i` to a `Vector` of length `o`.

```
linearLayer :: forall i o. (KnownNat i, KnownNat o) => Layer i o
```

Creates a *linear* `Layer`, i.e. a layer that multiplies the input with a weight `Matrix` and adds a bias to get the output.

Random initialization follows the recommendation from chapter 3 of the online book Neural Networks and Deep Learning by Michael Nielsen.

```
layer :: (KnownNat i, KnownNat o) => Diff' -> Layer i o
```

Creates a `Layer` as a combination of a linear layer and a non-linear activation function.

Layers

```
tanhLayer :: (KnownNat i, KnownNat o) => Layer i o
```

This is a simple `Layer`, specialized to `tanh`-activation. Output values are all in the interval [-1,1].

```
tanhLayer' :: (KnownNat i, KnownNat o) => Layer i o
```

This is a simple `Layer`, specialized to a modified `tanh`-activation, following the suggestion from Efficient BackProp by LeCun et al., where output values are all in the interval [-1.7159,1.7159].

```
logisticLayer :: (KnownNat i, KnownNat o) => Layer i o
```

This is a simple `Layer`, specialized to the logistic function as activation. Output values are all in the interval [0,1].

```
reLULayer :: (KnownNat i, KnownNat o) => Layer i o
```

This is a simple `Layer`, specialized to the *rectified linear unit* activation function. Output values are all non-negative.

Plumbing with Pipes

Pipes API

Modular and flexible components with “built in” gradient descent are nice,

but not enough: ...

Pipes API

Modular and flexible components with “built in” gradient descent are nice,

but not enough:

- Training “batches” must be provided - for realistic problems, not all training data will fit into memory.

Pipes API

Modular and flexible components with “built in” gradient descent are nice,

but not enough:

- Training “batches” must be provided - for realistic problems, not all training data will fit into memory.

- We want some progress reporting to see how training is going.

Pipes API

Modular and flexible components with “built in” gradient descent are nice,

but not enough:

- Training “batches” must be provided - for realistic problems, not all training data will fit into memory.
- We want some progress reporting to see how training is going.
- We want to stop training when some target is reached or no further improvement seems likely.

Pipes API

This can easily lead to a mess:

Pipes API

This can easily lead to a mess:

- IO is needed during training to load training data.

Pipes API

This can easily lead to a mess:

- IO is needed during training to load training data.

- Progress reporting has to be mixed with training code.

Pipes API

This can easily lead to a mess:

- IO is needed during training to load training data.
- Progress reporting has to be mixed with training code.
- Stop-criteria code has to be added to the mix.

Pipes API

This can easily lead to a mess:

- IO is needed during training to load training data.
- Progress reporting has to be mixed with training code.
- Stop-criteria code has to be added to the mix.

All this tends to produce imperative and non modular code…

Pipes API

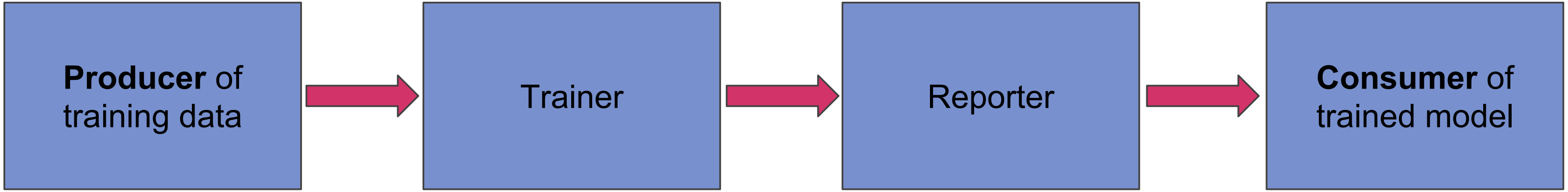
This can easily lead to a mess:

- IO is needed during training to load training data.
- Progress reporting has to be mixed with training code.
- Stop-criteria code has to be added to the mix.

All this tends to produce imperative and non modular code…

Pipes to the rescue!

Pipes API


Producer of training data
Trainer
Reporter
Consumer of trained model

Pipes API

```
data TS f g a b c
```

Source

Extensions BangPatterns

The training state of a model.

Constructors

TS

`tsModel :: Model f g a b c`	updated model
`tsGeneration :: Int`	generation
`tsEta :: Double`	learning rate
`tsBatchError :: Double`	last training error

```
descentP
```

Sourc

`:: (Foldable h, Monad m)`	
`=> Model f g a b c`	initial model
`-> Int`	first generation
`-> (Int -> Double)`	computes the learning rate from the generation
`-> Pipe (h a) (TS f g a b c) m r`	

A Pipe for training a model: It consumes mini-batches of samples from upstream and pushes the updated training state downstream.

Pipes API

simpleBatchP # Source

`:: MonadRandom m`	
`=> [a]`	all available samples
`-> Int`	mini-batch size
`-> Producer [a] m r`	

A simple `Producer` of mini-batches.

cachingBatchP # Source

`:: MonadRandom m`	
`=> ([Int] -> m [a])`	get samples with specified indices
`-> Int`	number of all available samples
`-> Int`	mini-batch size
`-> Int`	cache size
`-> Int`	number of cache reuses
`-> Producer [a] m s`	

Function `simpleBatchP` only works when all available samples fit into memory. If this is not the case, `cachingBatchP` can be used instead. It takes an effectful way to get specific samples and then caches some of those samples in memory for a couple of rounds, drawing mini-batches from the cached values.

Pipes API

reportTSP

```
:: Monad m
=> Int                                      report interval
-> (TS f g a b c -> m ())                   report action
-> Pipe (TS f g a b c) (TS f g a b c) m r
```

A `Pipe` for progress reporting of model training.

consumeTSP

```
:: Monad m
=> (TS f g a b c -> m (Maybe x))   check whether training is finished and what to return in that case
-> Consumer (TS f g a b c) m x
```

A `Consumer` of training states that decides when training is finished and then returns a value.

Other Features

Other Features

- already implemented
  - fixed-length vectors (used for layers)
  - support for *classification* problems (cross entropy,...)
  - normalization (input whitening)
- work in progress
  - convolutional networks
  - regularization
  - deep learning

Modules

- Data
  - Data.FixedSize fixed-size containers
    - Data.FixedSize.Class class definitions
    - Data.FixedSize.Matrix fixed-size matrices
    - Data.FixedSize.Vector fixed-length vectors
    - Data.FixedSize.Volume fixed-size volumes
  - Data.MyPrelude commonly used standard types and functions
  - Data.Utils various utilities
    - Data.Utils.Analytic "analytic" values
    - Data.Utils.Arrow arrow utilities
    - Data.Utils.Cache caching
    - Data.Utils.List list utilities
    - Data.Utils.Pipes pipe utilities
    - Data.Utils.Random random number utilities
    - Data.Utils.Stack a simple stack monad
    - Data.Utils.Statistics statistical utilities
    - Data.Utils.Traversable utilities for traversables
- Numeric
  - Numeric.Neural neural networks
    - Numeric.Neural.Convolution convolutional layers
    - Numeric.Neural.Layer layer components
    - Numeric.Neural.Model "neural" components and models
    - Numeric.Neural.Normalization normalizing data
    - Numeric.Neural.Pipes a pipes API for models

Examples

Sine Example

- approximate sin(x) on [0, 2π]

- 1 input neuron
- 4 hidden neurons
- 1 output neuron

- accurate to 0.1 after 850 generations

Sine Example

```haskell
main :: IO ()
main = flip evalRandT (mkStdGen 739570) $ do
    let xs = [0, 0.01 .. 2 * pi]
    m <- modelR $ whiten sinModel xs
    runEffect $
            simpleBatchP [(x, sin x) | x <- xs] 10
        >-> descentP m 1 (const 0.5)
        >-> reportTSP 50 report
        >-> consumeTSP check

  where

    sinModel :: StdModel (Vector 1) (Vector 1) Double Double
    sinModel = mkStdModel
        (tanhLayer . (tanhLayer :: Layer 1 4))
        (\x -> Diff $ Identity . sqDiff (pure $ fromDouble x))
        pure
        vhead
```

```haskell
getError ts =
    let m = tsModel ts
    in  maximum [abs (sin x - model m x) | x <- [0, 0.1 .. 2 * pi]]

report ts = liftIO $ do
    ANSI.clearScreen
    ANSI.setSGR [ANSI.SetColor ANSI.Foreground ANSI.Vivid ANSI.Red]
    ANSI.setCursorPosition 0 0
    printf "Generation %d\n" (tsGeneration ts)
    ANSI.setSGR [ANSI.Reset]
    graph (model (tsModel ts)) 0 (2 * pi) 20 50

check ts = return $ if getError ts < 0.1 then Just () else Nothing
```

Other Examples

- sqrt models the similar regression problem of approximating the square root function on the interval [0,4].
- iris solves the famous Iris Flower classification problem.
- MNIST tackles the equally famous MNIST problem of recognizing handwritten digits.

Questions? Comments?