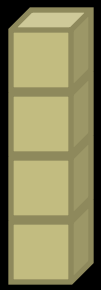
Typesafe Abstractions for Tensor Operations

Tongfei Chen
Johns Hopkins University

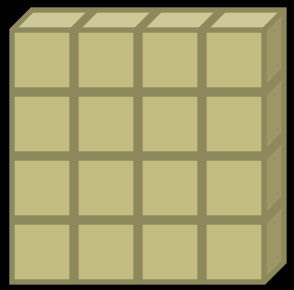
Tensors / NdArrays

Scalar (0-tensor)

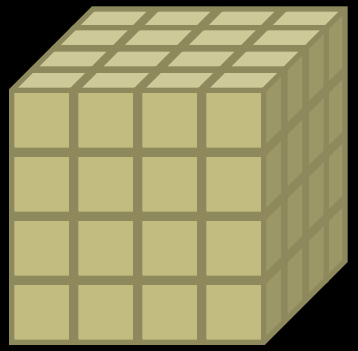
Vector (1-tensor)



Matrix (2-tensor)



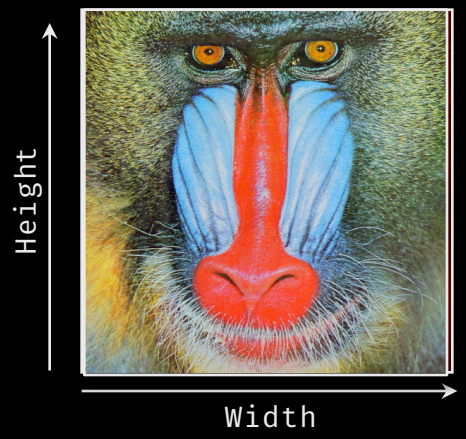
3-tensor



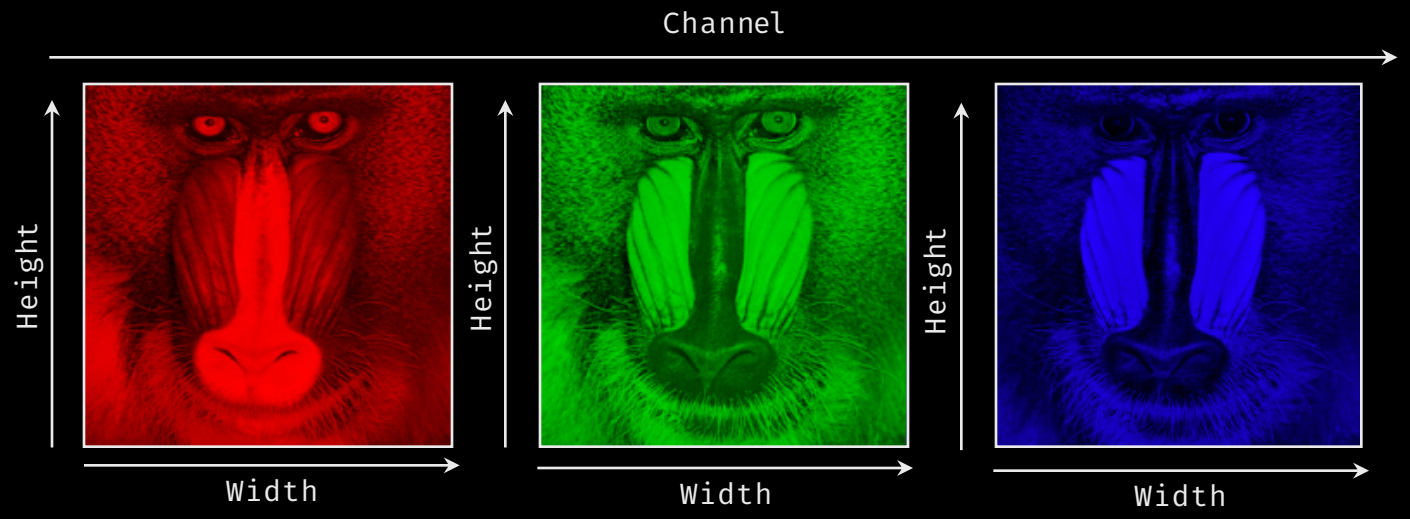
Tensors in ML

- Core data structure in machine learning
  - Computer vision (CV)
  - Natural language processing (NLP)
  - ...
- Especially useful in deep learning
  - Automatic differentiation

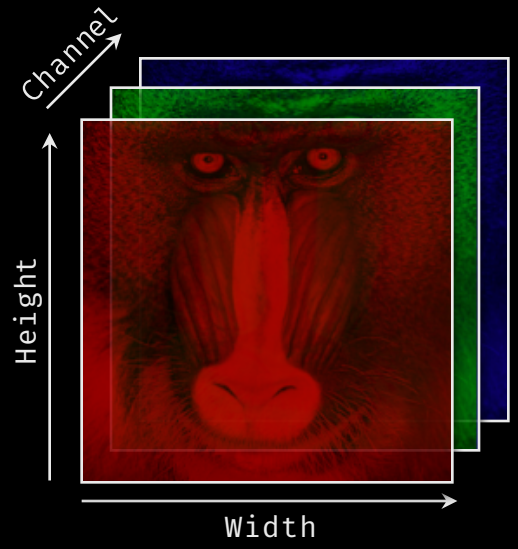
Images as tensors



Images as tensors

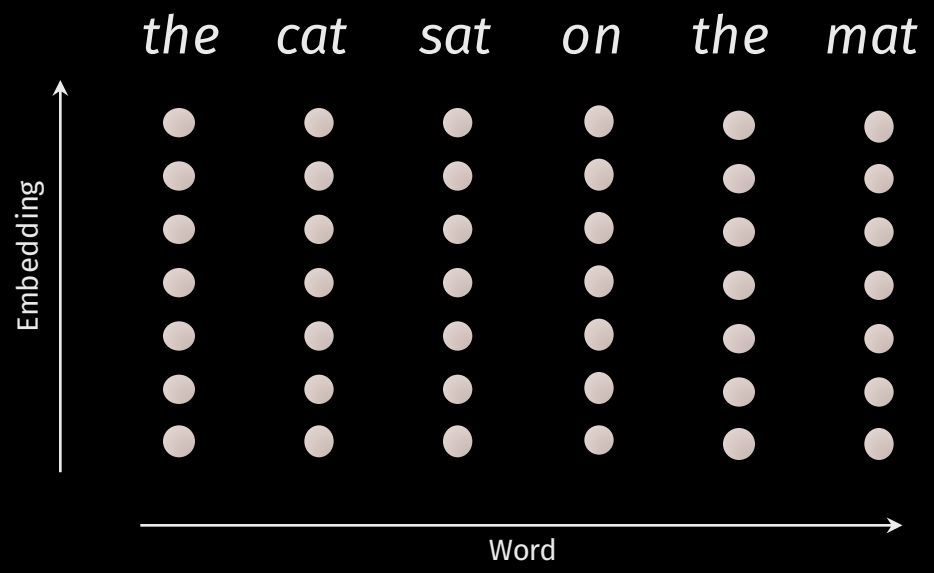


Images as tensors

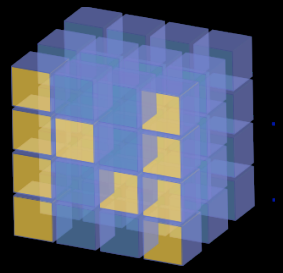


Sentences as tensors

$$embedding : V \to \mathbb{R}^d$$



ML tensor libraries



NumPy

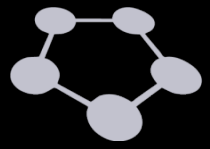
theano

TensorFlow ™

PYTORCH

∂y/net

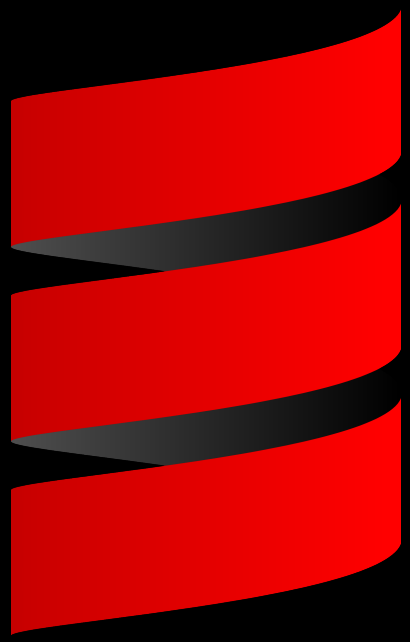
mxnet



Chainer


python


on



Lack of expressive types

- Only one type
  - Image: Tensor
  - Sentence: Tensor
- Do not know what each axis is
  - `reduce_max(a, axis=1)? # What is axis 1?`
- No type checking at compile time
  - Failure at runtime

Typefulness & Typesafety

- Something we sorely miss in Python libraries

- Can we make tensors typeful & typesafe?
- Encode meaning of axes

- HLists!
  - `Tensor[A <: HList]`

- Goal:
  - **Typeful & typesafe deep learning in Scala!**

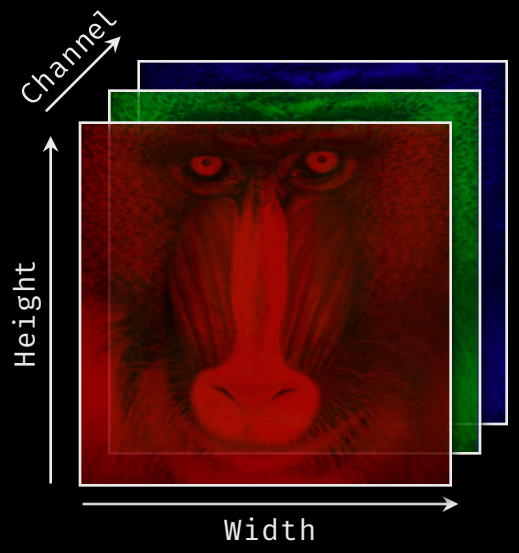
Typefulness

- `image = array([[...], …, ])`
- `image[w, h, c]`

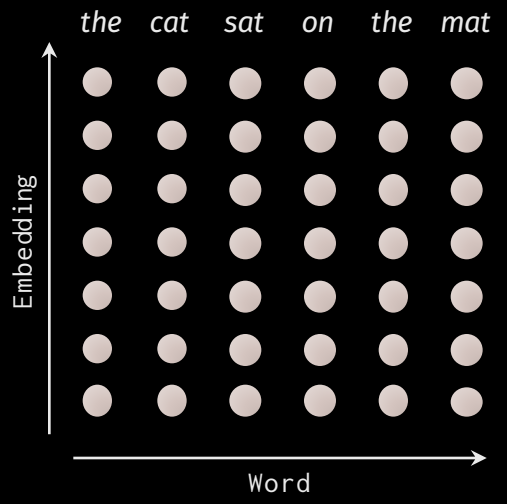
- `val image:`
- `Tensor[Width :: Height :: Channel :: HNil]`

Typefulness

- `Tensor[Width :: Height :: Channel :: HNil]`



- `Tensor[Word :: Embedding :: HNil]`



Typesafety

- Operations on tensors only allowed if their operand's axes make sense mathematically.

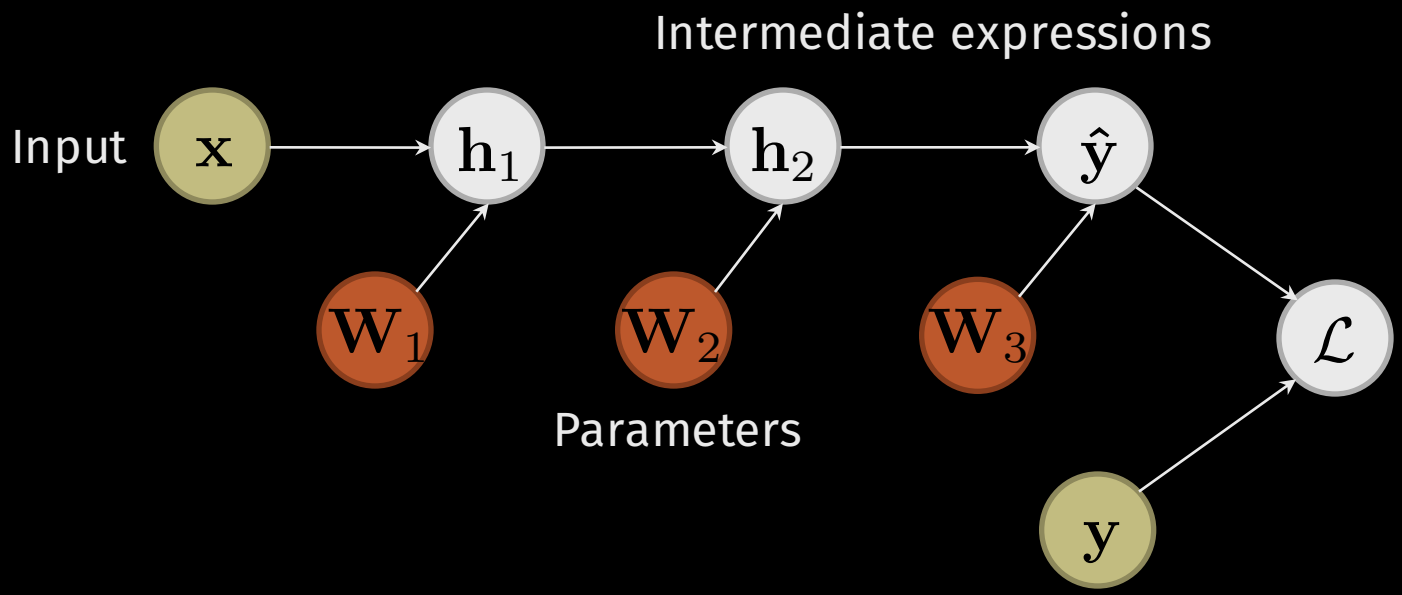
- ✅ `Tensor[A::$] + Tensor[A::$]`
- ❎ `Tensor[A::$] + Tensor[A::B::$]`
- ❎ `Tensor[A::$] + Tensor[B::$]`

Typesafety

$$\mathbf{Z} = \mathbf{X}\mathbf{Y}$$

- ❎ `MatMul(Tensor[A::$], Tensor[A::$])`
- ❎ `MatMul(Tensor[A::B::$], Tensor[A::B::$])`
- ✅ `MatMul(Tensor[A::B::$], Tensor[B::C::$])`

Computation graph



Computation graph

- `sealed trait Expr[X]`

GADT!

- `case class Input[X]() extends Expr[X]`
- `case class Param[X](var value: X) extends DExpr[X]`
- `case class Const[X](value: X) extends Expr[X]`

- ```
  case class Apply1[X, Y]
  (op: Op1[X, Y], x: Expr[X]) extends Expr[Y]
  ```

- ```
  case class Apply2[X1, X2, Y]
  (op: Op2[X1, X2, Y], x1: Expr[X1], x2: Expr[X2]) extends Expr[Y]
  ```

- ```
  case class Apply3[X1, X2, X3, Y]
  (op: Op3[X1, X2, X3, Y], x1: Expr[X1], x2: Expr[X2], x3: Expr[X3])
  extends Expr[Y]
  ```
  ```

Typesafe differentiable operators

$$\nabla_x \mathcal{L} = \nabla_y \mathcal{L} \cdot \frac{\partial y}{\partial x}$$

```
trait Op1[X, Y] {

  /** Name of this operation. */
  def name: String

  /** Applies this operation to a symbolic expression. */
  def apply(x: Expr[X]): Expr[Y] = Apply1(this, x)

  /** Applies this operation to a concrete value (forward computation). */
  def forward(x: X): Y

  /** Performs gradient backpropagation. */
  def backward(dy: Y, y: Y, x: X): X

}
```

Typesafe differentiable operators

$$\nabla_{x_1}\mathcal{L} = \nabla_y\mathcal{L} \cdot \frac{\partial y}{\partial x_1} \qquad \nabla_{x_2}\mathcal{L} = \nabla_y\mathcal{L} \cdot \frac{\partial y}{\partial x_2}$$

```
trait Op2[X1, X2, Y] {

  def name: String

  def apply(x1: Expr[X1], x2: Expr[X2]) = Apply2(this, x1, x2)

  def forward(x1: X1, x2: X2): Y

  def backward1(dy: Y, y: Y, x1: X1, x2: X2): X1

  def backward2(dy: Y, y: Y, x1: X1, x2: X2): X2

}
```

Type polymorphism

Poly1

Case.Aux

```
trait PolyOp1[F[X, Y] <: Op1[X, Y]] {
  /** Applies this operation to a concrete value (forward computation).
*/
  def apply[X, Y](x: X)(implicit f: F[X, Y]): Y = f.forward(x)

  /** Applies this operation to a symbolic expression. */
  def apply[X, Y](x: Expr[X])(implicit f: F[X, Y]): Expr[Y] =
    Apply1(f, x)
}
```

- Operation `PolyOp1[F]` can only be applied to `Expr[X]` if an implicit instance of `F[X, Y]` is found. If found, resulting type is `Expr[Y]`.
- `F[X, Y]` witnesses the fact that an operation `PolyOp1[F]` can turn `X` into `Y`. Applying the operation to `Expr[X]` requires proof `F[X, Y]`.

Examples: Negation

```scala
@implicitNotFound("Cannot apply Neg to ${X}.")
trait NegF[X, Y] extends DOp1[X, Y] {
  def name = "Neg"
}

object NegF {
  implicit def scalar[R](implicit R: IsReal[R]) =
    new NegF[R, R] {
      def tag = R
      def backward(dy: R, y: R, x: R) = -dy
      def forward(x: R) = -x
    }
  implicit def tensor[T[_ <: $$], R, A <: $$](implicit T: IsTypedRealTensor[T, R]) =
    new NegF[T[A], T[A]] {
      def tag = T.ground[A]
      def forward(x: T[A]) = -x
      def backward(dy: T[A], y: T[A], x: T[A]) = -dy
    }
}

object Neg extends PolyDOp1[NegF]
```

$\forall R, \texttt{IsReal}[R] \rightarrow \texttt{NegF}[R, R]$

$\forall T, R, A, \texttt{IsTypedRealTensor}[T, R] \rightarrow \texttt{NegF}[T[A], T[A]]$

```scala
enum NegF[X, Y] extends DOp1[X, Y] {

  implicit case Scalar[R](implicit R: IsReal[R])
    extends NegF[R, R]

  implicit case Tensor[T[_ <: $$, R, A <: $$]]
    (implicit T: IsTypedRealTensor[T, R])
    extends NegF[T[A], T[A]]

}
```

Examples: Addition

```scala
@implicitNotFound("Cannot  apply Add to ${X1} and ${X2}.")
trait AddF[X1, X2, Y] extends DOp2[X1, X2, Y] {
  def name = "Add"
}

object AddF {
```

$\forall R, \texttt{IsReal}[R] \rightarrow \texttt{AddF}[R, R, R]$

```scala
  implicit def scalar[R](implicit R: IsReal[R]) =
    new AddF[R, R, R] {
      def tag = R
      def forward(x1: R, x2: R) = add(x1, x2)
      def backward1(dy: R, y: R, x1: R, x2: R) = dy
      def backward2(dy: R, y: R, x1: R, x2: R) = dy
    }
```

$\forall T, R, A, \texttt{IsTypedRealTensor}[T, R] \rightarrow \texttt{AddF}[T[A], T[A], T[A]]$

```scala
  implicit def tensor[T[_ <: HList], R, A <: HList](implicit T: IsTypedRealTensor[T,
R]) =
    new AddF[T[A], T[A], T[A]] {
      def tag = T.ground[A]
      def forward(x1: T[A], x2: T[A]) = x1 + x2
      def backward1(dy: T[A], y: T[A], x1: T[A], x2: T[A]) = dy
      def backward2(dy: T[A], y: T[A], x1: T[A], x2: T[A]) = dy
    }
}

object Add extends PolyDOp2[AddF]
```

Examples: Matrix multiplication

```
@implicitNotFound("Cannot apply MatMul to ${X1} and ${X2}.")
trait MatMulF[X1, X2, Y] extends DOp2[X1, X2, Y] {
  def name = "MatMul"
}

object MatMulF {
```

$$\forall T, R, A, B, C, \texttt{IsTypedRealTensor}[T, R] \rightarrow \texttt{MatMulF}[T[AB], T[BC], T[AC]]$$

```
  implicit def matrix[T[_ <: $$], R, A, B, C](implicit T: IsTypedRealTensor[T, R]) =
    new MatMulF[T[A::B::$], T[B::C::$], T[A::C::$]] {
      import T._
      def tag = T.ground[A::C::$]
      def forward(x1: T[A::B::$], x2: T[B::C::$]) = mmMul(x1, x2)
      def backward1(dy: T[A::C::$], y: T[A::C::$], x1: T[A::B::$], x2: T[B::C::$]) =
        mmMul(dy, transpose(x2))
      def backward2(dy: T[A::C::$], y: T[A::C::$], x1: T[A::B::$], x2: T[B::C::$]) =
        mmMul(transpose(x1), dy)
    }
}

object MatMul extends PolyDOp2[MatMulF]
```

Example: Sum reduction

$$B_{ik} = \sum_j A_{ijk}$$

```
trait SumAlongF[U, X, Y] extends (U => DOp1[X, Y])

object SumAlongF {
```

$$\forall T, R, A, U, B, \left\{ \begin{array}{l} \texttt{IsTypedRealTensor}[T, R] \\ A \setminus U = B \end{array} \right\} \rightarrow \texttt{SumAlongF}[U, T[A], T[B]]$$

```
  implicit def tensor[T[_ <: $$], R, A <: $$, U, B <: $$]
  (implicit r: Remove.Aux[A, U, (U, B)], T: IsTypedRealTensor[T, R]) =
    new SumAlongF[U, T[A], T[B]] {
      import T._
      def apply(u: U) = new DOp1[T[A], T[B]] {
      def tag = T.ground[B]
      def name = s"SumAlong[$u]"
      def forward(x: T[A]) = sumAlong(x, r.index)
      def backward(dy: T[B], y: T[B], x: T[A]) = broadcast(dy, r.removed,
r.index)
    }
  }

}

case class SumAlong[U](parameter: U) extends ParaPolyDOp1[U, SumAlongF]
```

Contraction

$$C_{i_1,\cdots,i_m,k_1,\cdots,k_p} = \sum_{j_1,\cdots,j_n} A_{i_1,\cdots,i_m,j_1,\cdots,j_n} B_{j_1,\cdots,j_n,k_1,\cdots,k_p}$$

- Tensordot / einsum (Einstein summation)
- Subsumes many common operators

Inner product	$C = \sum_i A_i B_i$
Matrix multiplication	$C_{ik} = \sum_j A_{ij} B_{jk}$
Tensor product	$C_{i_1,\cdots,i_m,j_1,\cdots,j_n} = A_{i_1,\cdots,i_m} B_{j_1,\cdots,j_n}$

Natural contraction

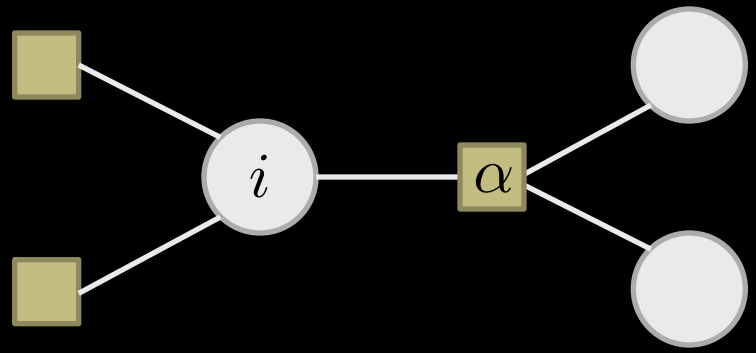
- Contract those axes with the same type tags!
  - Akin to database (relational algebra) inner join ($\bowtie$)
  - Join those columns with the same names

- Given `Tensor[A]` and `Tensor[B]`, output `Tensor[C]`
  - where C = A $\triangle$ B (symmetric difference)

- Widely used in ML (e.g. belief propagation):

$$m_{i\to\alpha} = \bigodot_{\beta\in N(i)\setminus\{\alpha\}} m_{\beta\to i}$$

$$m_{\alpha\to i} = \psi_\alpha \bowtie \bigbowtie_{j\in N(\alpha)\setminus\{i\}} m_{j\to\alpha}$$



Example: Contraction

```
@implicitNotFound("Cannot  apply Contract to ${X1} and ${X2}.")
trait ContractF[X1, X2, Y] extends DOp2[X1, X2, Y] {
    def name = "Contract"
}

object ContractF {
```

$$\forall T, R, A, B, C, \left\{ \begin{array}{l} \texttt{IsTypedRealTensor}[T, R] \\ A \triangle B = C \end{array} \right\} \rightarrow \texttt{ContractF}[T[A], T[B], T[C]]$$

```
  implicit def tensor[T[_ <: $$], R, A <: $$, B <: $$, C <: $$](implicit T:
IsTypedRealTensor[T, R], sd: SymDiff.Aux[A, B, C]) =
        new ContractF[T[A], T[B], T[C]] {
            import T._
            def tag = T.ground[C]
            def forward(x1: T[A], x2: T[B]) = contract(x1, x2)(sd)
            def backward1(dy: T[C], y: T[C], x1: T[A], x2: T[B]) = contract(dy,
x2)(sd.recoverLeft)
            def backward2(dy: T[C], y: T[C], x1: T[A], x2: T[B]) = contract(dy,
x1)(sd.recoverRight)
        }
}

object Contract extends PolyDOp2[ContractF]
```

Typelevel symmetric difference

```
trait SymDiff[A <: HList, B <: HList] extends DepFn2[A, B] {
  type Out <: HList
  def matchedIndices: List[(Int, Int)]
  def lhsRetainedIndices: List[(Int, Int)]
  def rhsRetainedIndices: List[(Int, Int)]
  def recoverLeft: SymDiff.Aux[Out, B, A]
  def recoverRight: SymDiff.Aux[Out, A, B]
}

object SymDiff {

  def apply[A <: HList, B <: HList](implicit o: SymDiff[A, B]): Aux[A, B, o.Out] = o
  type Aux[A <: HList, B <: HList, C <: HList] = SymDiff[A, B] { type Out = C }

  // A =:= HNil
  implicit def case0[B <: HList, N <: Nat](implicit bl: Length.Aux[B, N], bln:
ToInt[N]): Aux[HNil, B, B] = ???

  // A.head ∉ B => A.head ∈ C
  implicit def case1[H, T <: HList, B <: HList, C <: HList]
  (implicit n: NotContains[B, H], s: SymDiff.Aux[T, B, C]): Aux[H :: T, B, H :: C]
= ???

  // A.head ∈ B => A.head ∉ C
  implicit def case2[H, T <: HList, B <: HList, R <: HList, N <: Nat, C <: HList]
  (implicit i: IndexOf.Aux[B, H, N], r: RemoveAt.Aux[B, N, R], s: SymDiff.Aux[T, R,
C]): Aux[H :: T, B, C] = ???

}
```

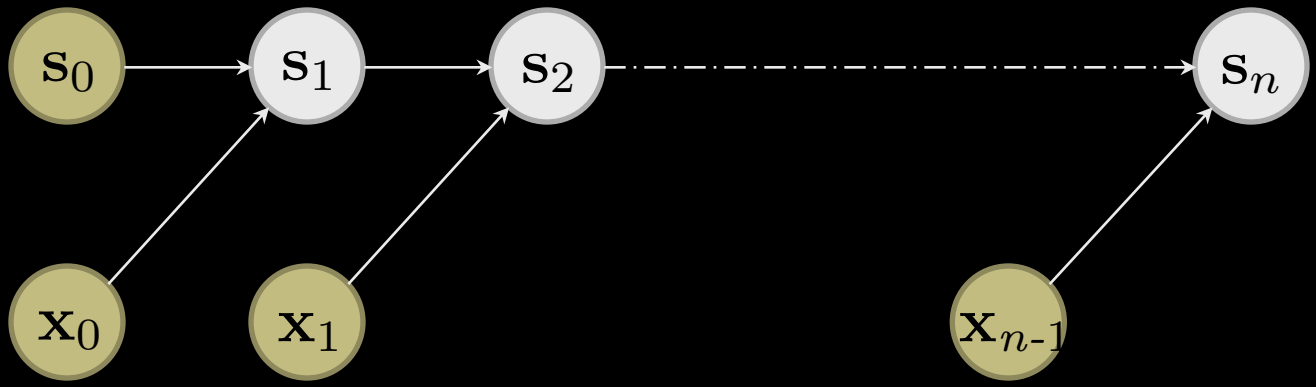
Elegant property

$$A \bowtie B = C$$

$$\nabla A = \nabla C \bowtie B \qquad \nabla B = \nabla C \bowtie A$$

$$A = (A \triangle B) \triangle B \qquad B = (A \triangle B) \triangle A$$

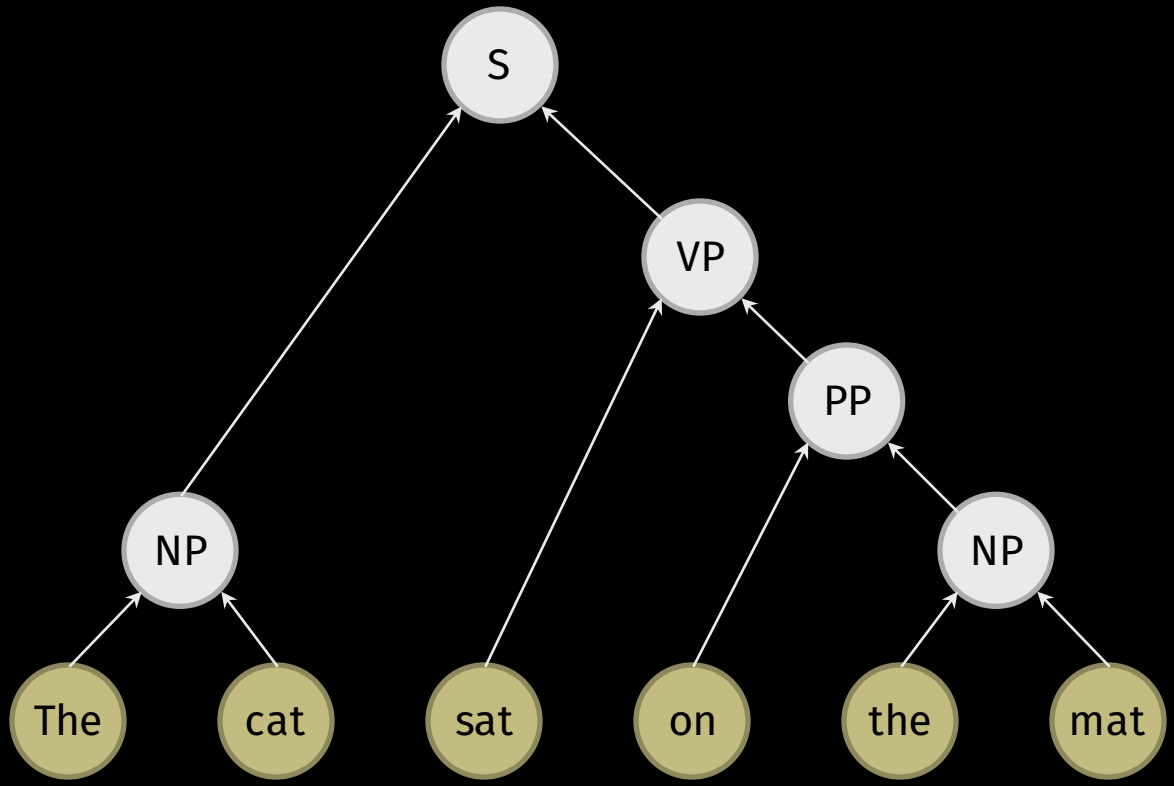
Recurrent neural network




```
• type RecurrentUnit[S, I] =
((Expr[S], Expr[I]) => Expr[S])
```

- fold / scan / etc.

Recursive neural network



- `type BinaryTreeRecurrentUnit[S, I] = ((Expr[S], Expr[S], Expr[I]) => Expr[S])`
- Generic fold (catamorphisms)

Example network

```
val x = Input[DenseTensor[In::$]]()
val y = Input[DenseTensor[Out::$]]()

val L1 = Affine(In -> 784, Hidden1 -> 300)
val L2 = Affine(Hidden1 -> 300, Hidden2 -> 100)
val L3 = Affine(Hidden2 -> 100, Out -> 10)

val ŷ =
  x |> L1 |> ReLU |> L2 |> ReLU |> L3 |> Softmax
val loss = CrossEntropy(y, ŷ)
```

Future work

- Towards a fully-fledged Scala deep learning engine
  - Automatic batching (fusion of computation graphs)
  - GPU support (through SWIG)
  - Multiple backends (Torch/MXNet/etc.)
  - Typesafe higher-order gradients
  - Distributed learning (through Spark)

- Goals
  - Typeful & typesafe
  - Declarative, succinct & highly expressive
  - Fast on GPUs

- https://github.com/ctongfei/nexus

Q & A