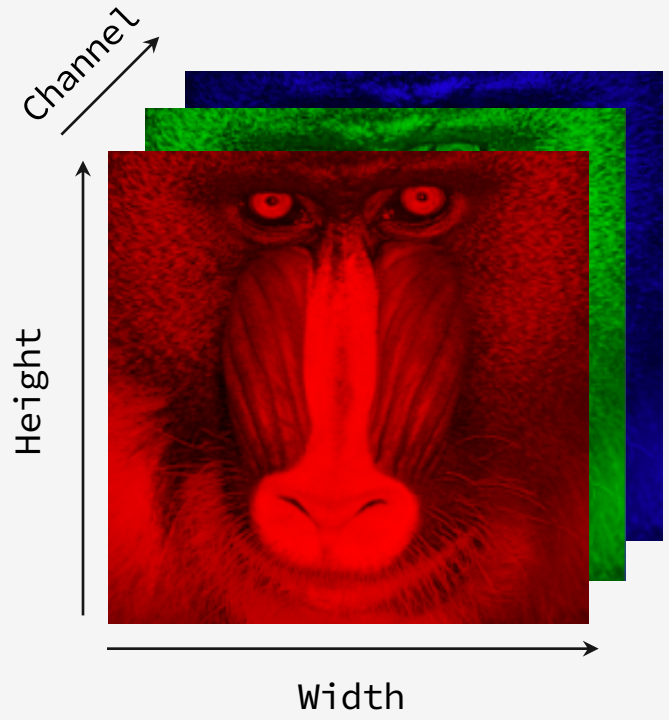
Towards **TypeSafe** Deep Learning in Scala

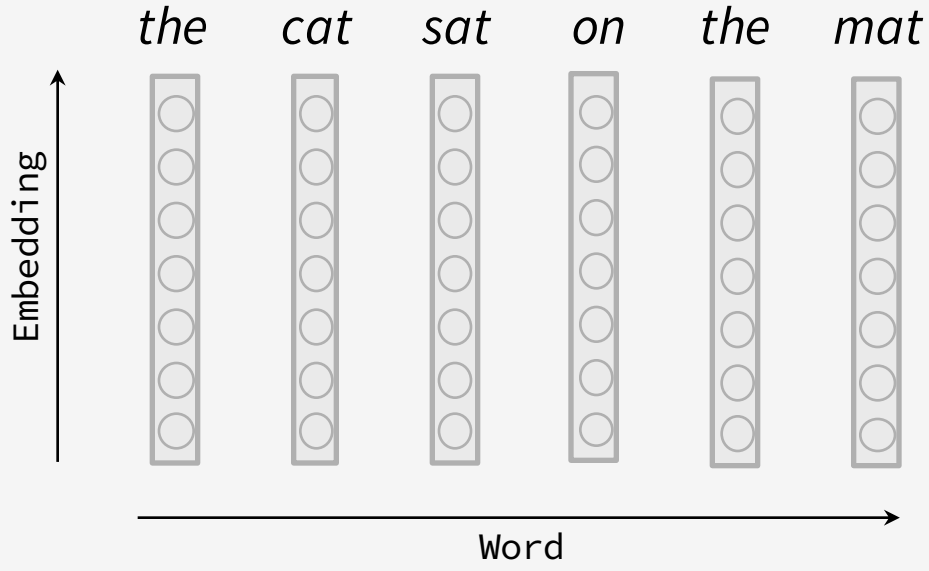
Tongfei Chen

Johns Hopkins University

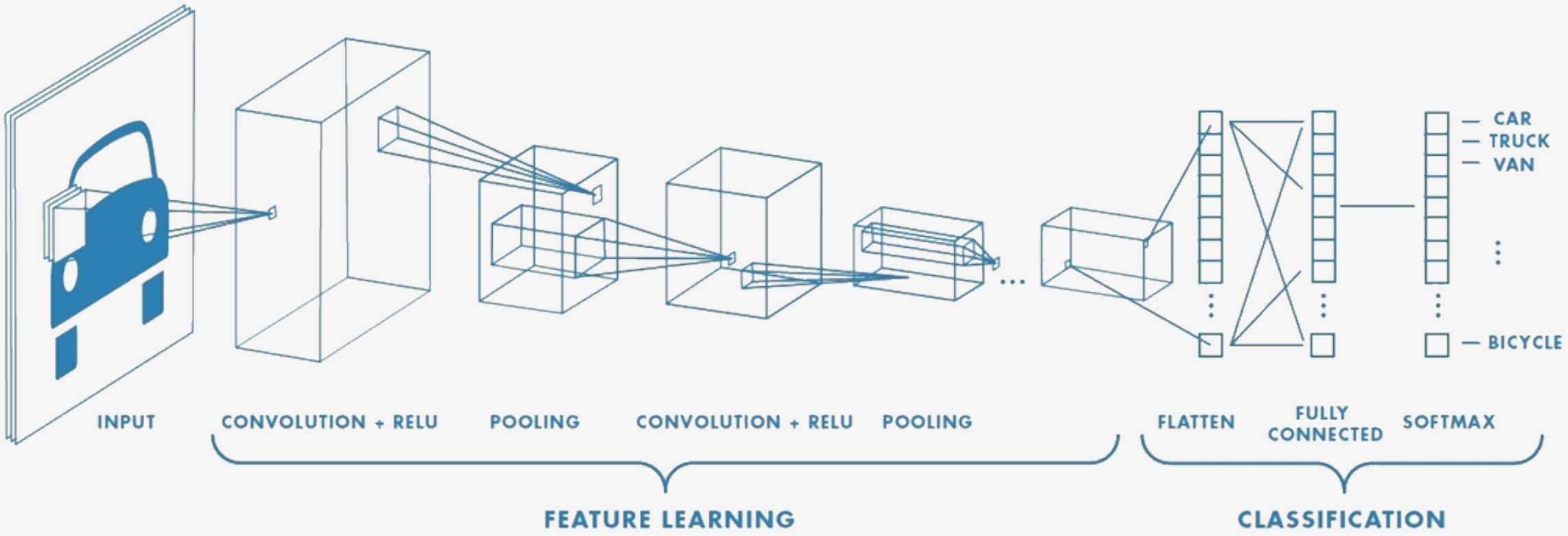
Deep learning in a nutshell

- Hype around AI
- Core data structure: Tensors
  - A.k.a. Multidimensional arrays (`NdArray`)





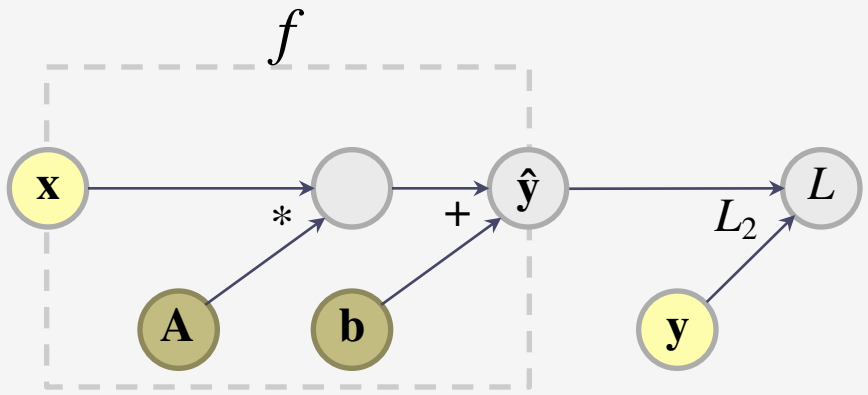
Deep learning in a nutshell



- Credits: MathWorks: https://www.mathworks.com/discovery/convolutional-neural-network.html

Deep learning in a nutshell

- Function fitting!
  - Linear regression:
  - $f : \mathbb{R}^m \rightarrow \mathbb{R}^n;\ \hat{\mathbf{y}} = \mathbf{A}\mathbf{x} + \mathbf{b}$
  - Machine translation:
  - $f : \mathrm{Fr} \rightarrow \mathrm{En}$



- Model (function to fit):
  - is composed from smaller building blocks with parameters;
  - trained by gradient descent with respect to a loss function.

$$L = \|\hat{\mathbf{y}} - \mathbf{y}\|^2$$

- *Deep Learning est mort. Vive Differentiable Programming!* (LeCun, 2018)

Common deep learning libraries

TensorFlow

PYTORCH

mxnet

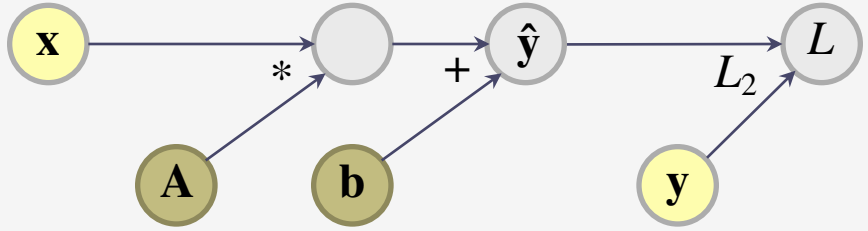
∂y/net

Caffe2

Chainer


on
TM

The Pythonic way (TensorFlow)



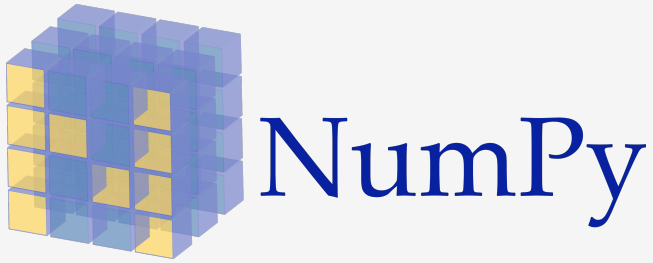

```
x = tf.placeholder(tf.float32, [m])
y = tf.placeholder(tf.float32, [n])

A = tf.Variable(tf.random_normal([n, m]))
b = tf.Variable(tf.random_normal([n]))

Ax = tf.multiply(x, A)
pred = tf.add(Ax, b)

cost = tf.reduce_sum(tf.pow(pred - y, 2))
```

A more complex example (PyTorch)

```
def forward(self, x: Variable) → Variable:
    """
    :param x: LongTensor[Batch, Word]
    :return: FloatTensor[Batch, Word, Embedding]
    """

    x_embedded = self.EmbeddingLayer(x).transpose(0, 1)
        # FloatTensor[Word, Batch, Emb]

    batch_size = x.size(0)
    h0_batched = self.h0.unsqueeze(1).expand(self.h0.size(0), batch_size, self.h0.size(1)).contiguous()
        # [Layer*Direction, Batch, Emb]
    c0_batched = self.c0.unsqueeze(1).expand(self.c0.size(0), batch_size, self.c0.size(1)).contiguous()
        # [Layer*Direction, Batch, Emb]

    output, _ = self.RecurrentLayer(x_embedded, (h0_batched, c0_batched))
        # [Word, Batch, Emb]

    return output.transpose(0, 1)
        # [Batch, Word, Emb]
```

The Pythonic approach

- Everything belongs to one type: `Tensor`
  - Vectors / Matrices
  - Sequence of vectors / Sequence of matrices
  - Images / Videos / Words / Sentences / …



- How many axes are in there? What does each axis stand for?

- Programmers track the axes and shape by themselves
  - Pythonistas can remember them by heart!
  - However, as a static typist, I cannot remember all these – I need types to guide me


nescala 2018
GIVE ME TYPE SAFETY OR GIVE ME DEATH

Nexus: **Typesafe** Deep Learning

https://github.com/ctongfei/nexus

Typesafe tensors: goal

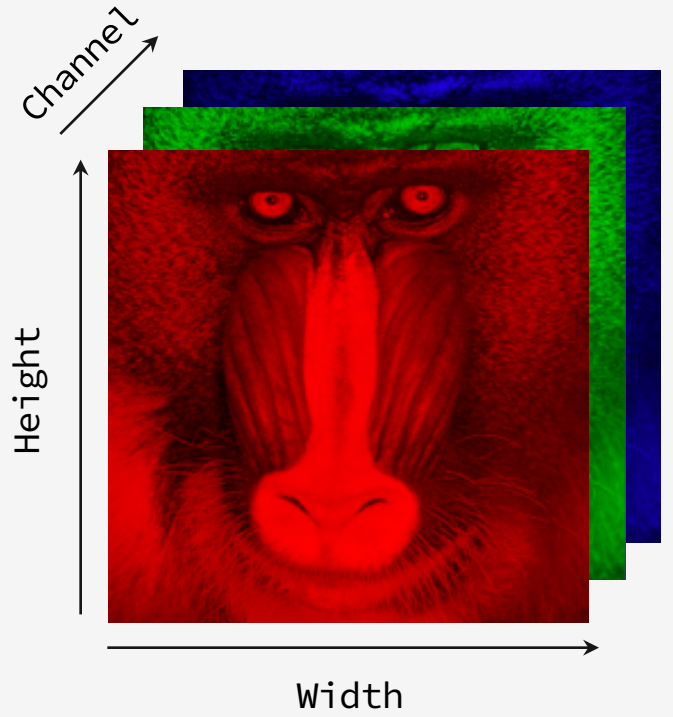
`Tensor[Axes]`

- "`Axes`" is the tensor axes descriptor – describes the semantics of each axis
  - A tuple of singleton types (labels to axes)

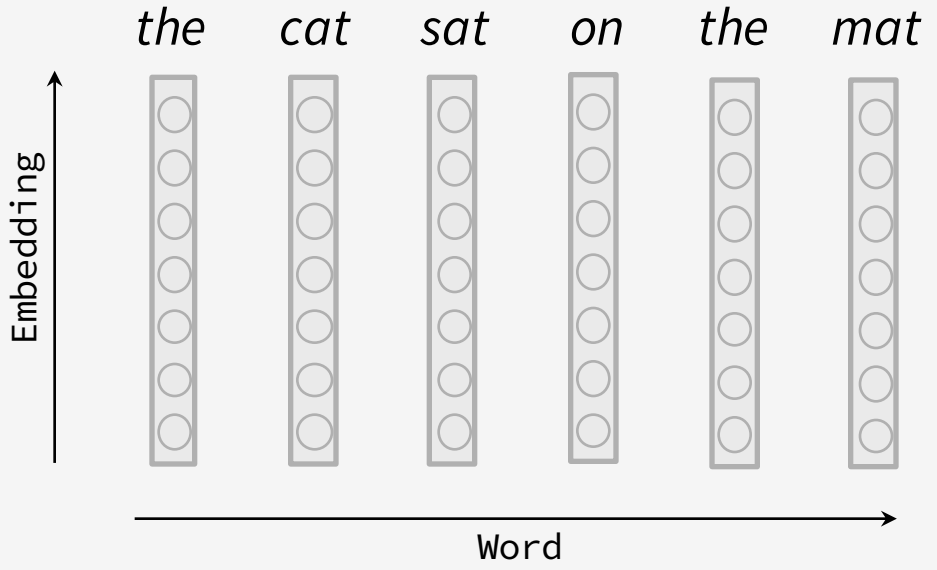
- All operations on tensors are statically typed
  - Result types known at compile time – IDE can help programmers
  - Compilation failure when operating incompatible tensors

Typesafe tensors

- `FloatTensor[(Width, Height, Channel)]`



- `FloatTensor[(Word, Embedding)]`



Typesafety guarantees

- 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 guarantees

- Matrix multiplication

- ❌ `MatMul(Tensor[A], Tensor[A])`
- ❌ `MatMul(Tensor[(A, B)], Tensor[(A, B)])`
- ✅ `MatMul(Tensor[(A, B)], Tensor[(B, C)])`

Typesafety guarantees

- Axis reduction operations

$$Y_{ik} = \sum_j X_{ijk}$$

- Python (TensorFlow): `tf.reduce_sum(X, dim=1)`

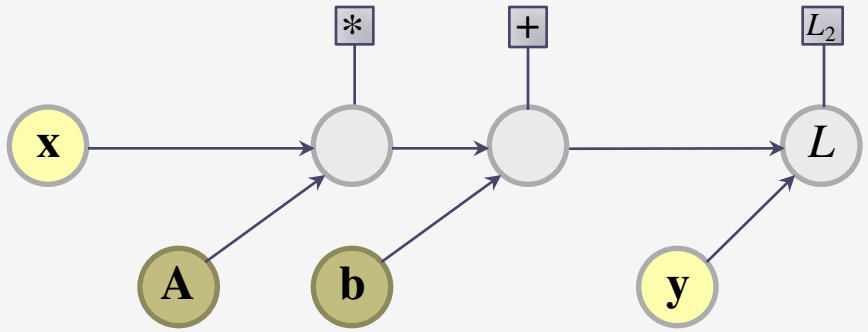
- `X: Tensor[(A, B, C)]`
- ✅ `SumAlong(B)(X): Tensor[(A, C)]`
- ❌ `SumAlong(D)(X)`

Tuples ⟺ HLists

- HLists are easier to manipulate
  - Underlying typelevel manipulation is done using HLists
- Use `Generic` and `Tupler` in Shapeless

- `Generic.Aux[A, B]` proves that the the HList form of A is B
- `Tupler.Aux[B, A]` proves that the tuple form of B is A

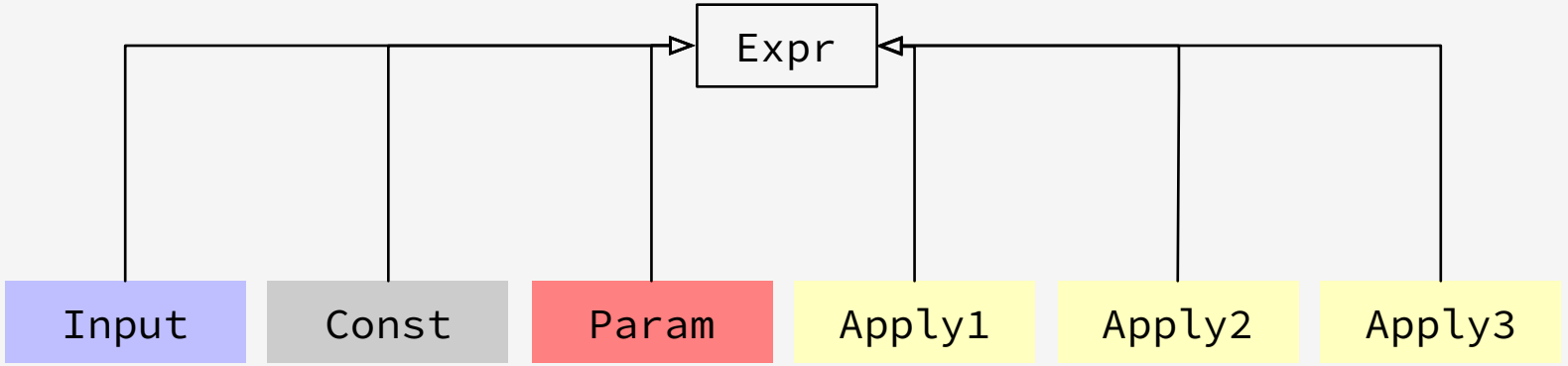
Typesafe computation graphs: GADTs



- `sealed trait Expr[X]`
- `case class Input[X] extends Expr[X]`
- `case class Param[X](var value: X)`
  `(implicit val tag: Grad[X]) extends Expr[X]`
- `case class Const[X](value: X) extends Expr[X]`

- `case class App1[X, Y](op: Op1[X, Y], x: Expr[X]) extends Expr[Y]`
- `case class App2[X1, X2, Y](op: Op2[X1, X2, Y], x1: Expr[X1],`
  `x2: Expr[X2]) extends Expr[Y]`

- ……



Typesafe differentiable operators

```
trait Op1[X, Y] extends Func1[X, Y]  {

  def apply(x: Expr[X]): Expr[Y] = App1(this, x)

  def forward(x: X): Y

  def backward(dy: Y, y: Y, x: X): X

}
```

$$y = f(x_1, x_2)$$

$$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial y}\frac{\partial y}{\partial x}$$

Typesafe differentiable operators

```
trait Op2[X1, X2, Y] extends Func2[X1, X2, Y] {

  def apply(x1: Expr[X1], x2: Expr[X2]) = App2(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

}
```

$$y = f(x_1, x_2)$$

$$\frac{\partial L}{\partial x_1} = \frac{\partial L}{\partial y}\frac{\partial y}{\partial x_1}$$

$$\frac{\partial L}{\partial x_2} = \frac{\partial L}{\partial y}\frac{\partial y}{\partial x_2}$$

Forward computation

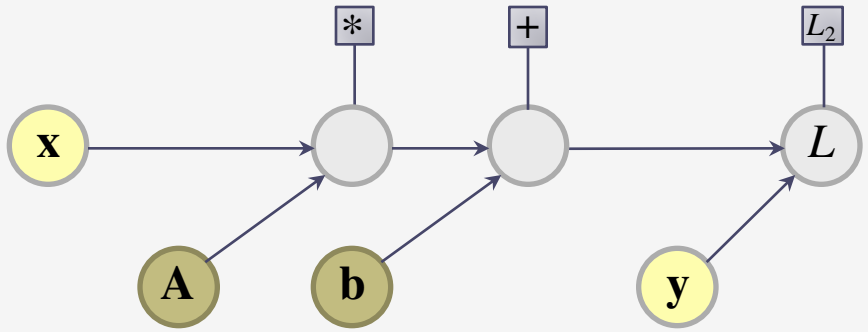
- Type:

```
Expr[A] => A
```

- With Cats:

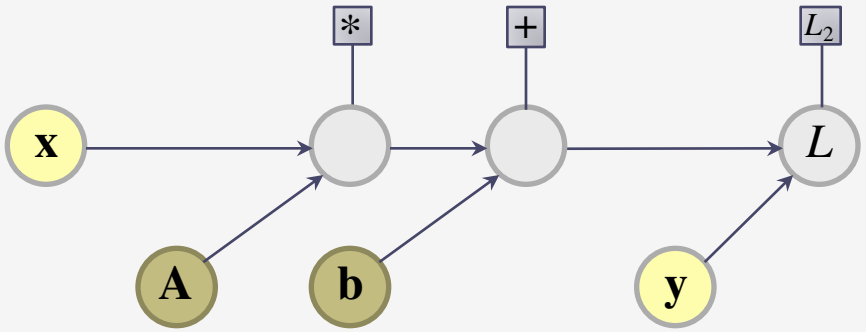
```
Expr ~> Id
```

- Interpreting the computation graph




```
def apply[A](e: Expr[A]): A = {
  if (values contains e) values(e)
  else e match {

    case Param(x, _) ⇒ values(e) = x; x
    case Const(x, _) ⇒ values(e) = x; x

    case App1(o, x) ⇒
      val y = o.forward(apply(x))
      values(e) = y; y
    case App2(o, x1, x2) ⇒
      val y = o.forward(apply(x1), apply(x2))
      values(e) = y; y
    case App3(o, x1, x2, x3) ⇒
      val y = o.forward(apply(x1), apply(x2), apply(x3))
      values(e) = y; y

    case e @ Input(_) ⇒
      throw new InputNotGivenException(e) // should already be in `values`
  }
}
```

Backward (gradient) computation



- From last node (loss), traverse the graph
  - Reversed ordering of forward computation
- For each node *x*, compute the gradient of the loss with respect to *x*

Operators vs modules

- Operators: Can be directly computed using the forward method
- Modules: Must use an interpreter to interpret (contains computation subgraph)

Supertype for all symbolic functions

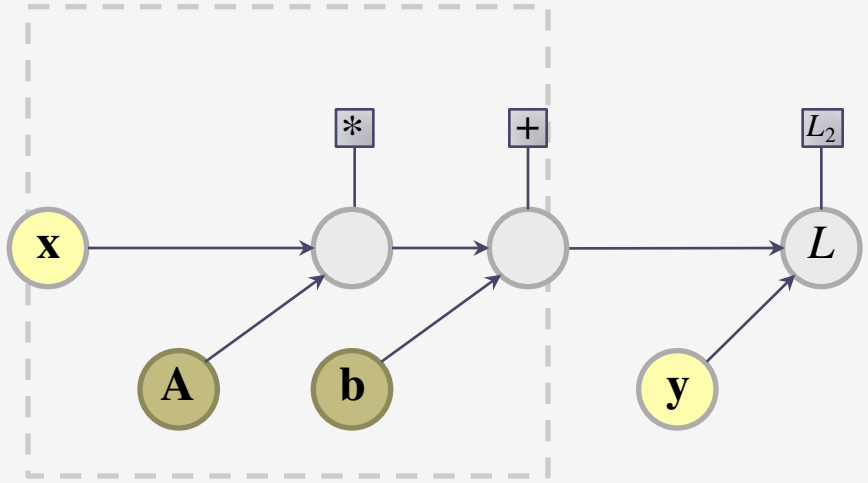
```
Func1[X, Y]   = (Expr[X] => Expr[Y])
```

```
Op1[X, Y]
```

```
forward(x: X): Y
backward(dy: Y, y: Y, x: X): X
```

```
Module1[X, Y]
```

```
parameters: Set[Param[_]]
```



Polymorphic symbolic functions

- `Op[X, Y]` only applies on one type: `X`
- We need type polymorphism. Similar to Shapeless's `Poly1`:


```
trait PolyFunc1 {

  type F[X, Y]

  def ground[X, Y](implicit f: F[X, Y]): Func1[X, Y]

  def apply[X, Y](x: Expr[X])(implicit f: F[X, Y]): Expr[Y] =
    ground(f)(x)
}
```

Polymorphic symbolic functions

```
def apply[X, Y](x: Expr[X])(implicit f: F[X, Y]): Expr[Y]
```

- Only applicable when `op.F[X, Y]` found. If found, result type is `Expr[Y]`.
- `F[_, _]` is an arbitrary typelevel predicate!
- `op.F[X, Y]` ⟺ op can be applied to `Expr[X]`, and it results in `Expr[Y]`.

- Compiling as proving (Curry-Howard correspondence!)
- Implicit `F[X, Y]` found ⟺ Proposition `F[X, Y]` proven
- We can encode any type constraint we want on type operators into F.

Polymorphic operators

```
abstract class PolyOp1 extends PolyFunc1 {

@implicitNotFound("This operator cannot be applied to an
argument of type ${X}.")
trait F[X, Y] extends Op1[X, Y]

def ground[X, Y](implicit f: F[X, Y]) = f

override def apply[X, Y](x: Expr[X])(implicit f: F[X, Y]) =
  f(x)

}
```

For polymorphic operators, the proof F is the grounded operator itself

Example: Add

- Two variables of the same type, and can be differentiated against can be added.

$$\forall X, \texttt{Grad}[X] \rightarrow \texttt{Add.F}[X, X, X]$$

```
object Add extends PolyOp2 {

  implicit def addF[X: Grad]: F[X, X, X] = new F[X, X, X] {
    def name = "Add"
    def tag(tx1: Type[X], tx2: Type[X]) = tx1
    def forward(x1: X, x2: X): X = x1 + x2
    def backward1(dy: X, y: X, x1: X, x2: X): X = dy
    def backward2(dy: X, y: X, x1: X, x2: X): X = dy
  }

}
```

Example: MatMul

- Two matrices can be multiplied when the second axis of the first matrix coincides with the first axis of the second matrix.

$$\forall T, R, A, B, C, \texttt{IsRealTensorK}[T, R] \rightarrow \texttt{MatMul.F}[T[A, B], T[B, C], T[A, C]]$$

```
object MatMul extends PolyOp2 {

  implicit def matMulF[T[_], R, A, B, C]
  (implicit T: IsRealTensorK[T, R]): F[T[(A, B)], T[(B, C)], T[(A, C)]] =
    new F[T[(A, B)], T[(B, C)], T[(A, C)]] {
      import T._
      def name = "MatMul"
      def tag(tx1: Type[T[(A, B)]], tx2: Type[T[(B, C)]]) = 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)
    }
}
```

Parameterized polymorphic operators

- Sometimes operators depend on parameters not part of the computation graph

```
abstract class ParameterizedPolyOp1 { self =>

  trait F[X, Y] extends Op1[X, Y]

  class Proxy[P](val parameter: P) extends PolyFunc1 {
    type F[X, Y] = P => self.F[X, Y]
    def ground[X, Y](implicit f: F[X, Y]) = f(parameter)
  }
  def apply[P](parameter: P): Proxy[P] = new Proxy(parameter)

}
```

Example: Axis renaming

- `Rename(A -> B)(x)`

$$\forall T, E, A, U, V, B, \left\{ \begin{array}{l} \texttt{IsTensorK}[T, E] \\ A \setminus \{U\} \cup \{V\} = B \end{array} \right\} \rightarrow \texttt{Rename.F}[T[A], T[B]]$$

```
object Rename extends ParameterizedPolyOp1 {

  implicit def renameF[T[_], E, A, U: Label, V: Label, B]
  (implicit r: Replace.Aux[A, U, V, B], T: IsTensorK[T, E]) = (uv: (U, V)) =>
    new F[T[A], T[B]] {
      val (u, v) = uv
      import T._
      def name = s"Rename[${typeName(u)} -> ${typeName(v)}]"
      def tag(tx: Type[T[A]]) = T.ground[B]
      def forward(x: T[A]) = typeWith[B](untype(x))
      def backward(dy: T[B], y: T[B], x: T[A]) = typeWith[A](untype(dy))
    }

}
```

Example: Sum along axis

$$Y_{ik} = \sum_j X_{ijk}$$

- `IndexOf.Aux[A, U, N]`: The N-th type of A is U
- `RemoveAt.Aux[A, N, B]`: A, with the N-th type removed, is B

$$\forall T, R, A, U, B, \left\{ \begin{array}{l} \texttt{IsRealTensorK}[T, R] \\ A \setminus \{U\} = B \end{array} \right\} \rightarrow \texttt{SumAlong.F}[T[A], T[B]]$$

```
object SumAlong extends ParameterizedPolyOp1 {

  implicit def sumAlongF[T[_], R, A, U: Label, N <: Nat, B]
    (implicit T: IsRealTensorK[T, R], ix: IndexOf.Aux[A, U, N], ra: RemoveAt.Aux[A, N, B]) = (u: U) ⇒
    new F[T[A], T[B]] {
      def name = s"SumAlong[${typeName(u)}]"
      def tag(tx: Type[T[A]]) = T.ground[B]
      def forward(x: T[A]) = T.sumAlong(x, ix.toInt)
      def backward(dy: T[B], y: T[B], x: T[A]) = T.expandDim(dy, ix.toInt, T.size(x, ix.toInt))
    }

}
```

IndexOf in the style of Shapeless

$$\texttt{IndexOf.Aux}[X :: T, X, \_0]$$

```
implicit def indexOfHListCase0[T <: HList, X]: Aux[X :: T, X, _0] =
  new IndexOf[X :: T, X] {
    type Out = _0
    def apply() = Nat._0
    def toInt = 0
  }
```

$$\texttt{IndexOf.Aux}[T, X, I] \rightarrow \texttt{IndexOf.Aux}[H :: T, X, I + 1]$$

```
implicit def indexOfHListCaseN[T <: HList, H, X, I <: Nat]
(implicit p: IndexOf.Aux[T, X, I]): Aux[H :: T, X, Succ[I]] =
  new IndexOf[H :: T, X] {
    type Out = Succ[I]
    def apply() = Succ[I]()
    def toInt = p.toInt + 1
  }
```

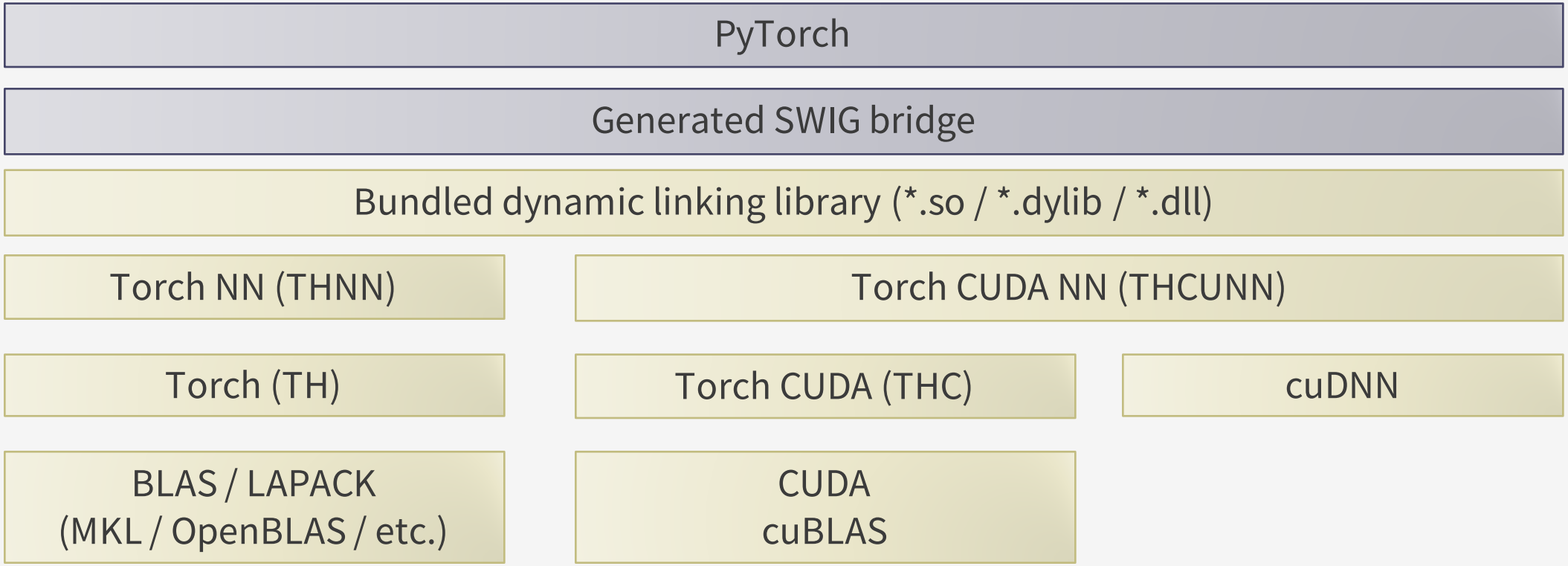
Native C / CUDA integration

- Doing math in JVM is not efficient
- Integration with native code through JNI
- Underlying C/C++ code; JNI code generated by SWIG

- Native CPU backend: BLAS/LAPACK from MKL/OpenBLAS/etc.
- CUDA GPU backend: cuBLAS/cuDNN
- OpenCL GPU backend?

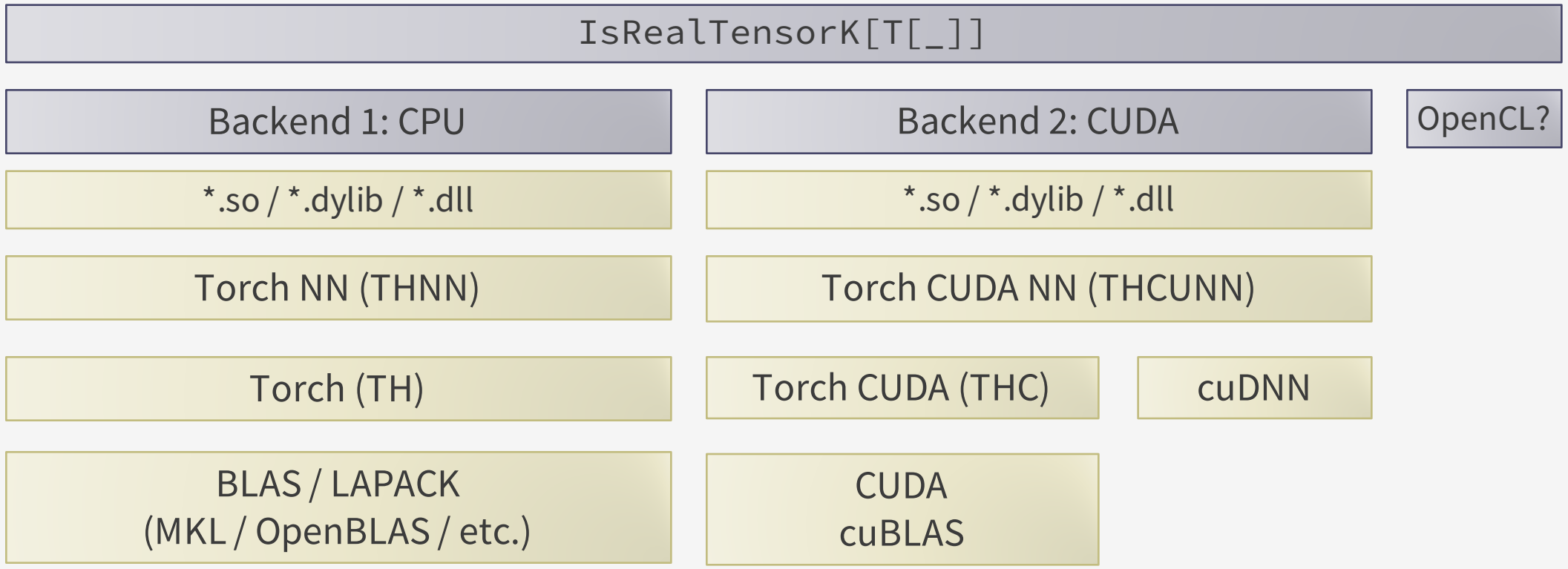
Example approach (PyTorch)

- Bridging Python with native CPU / CUDA code



Supporting multiple backends

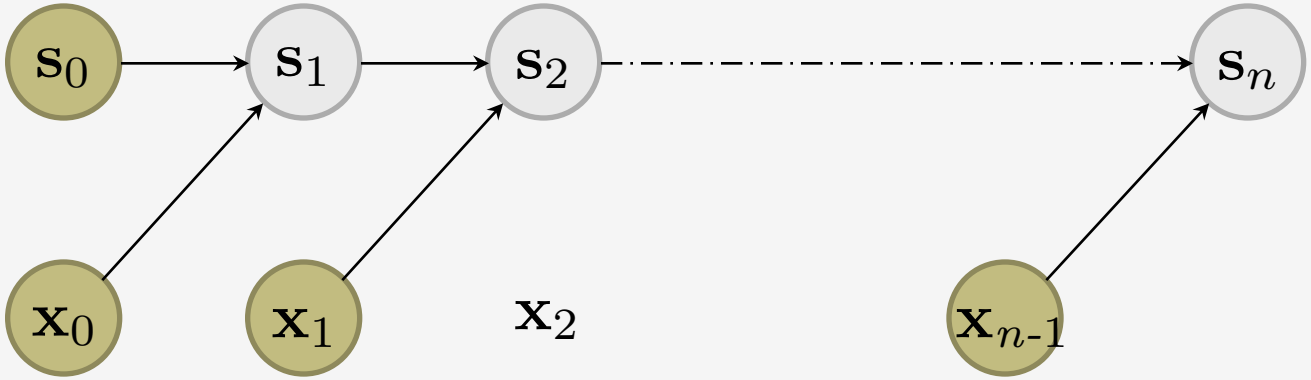
- Bridging JVM with native CPU / CUDA code through SWIG-generated JNI code
- Reusing C/C++ backends from existing libraries (PyTorch / etc.)



Neural networks with dynamic structures

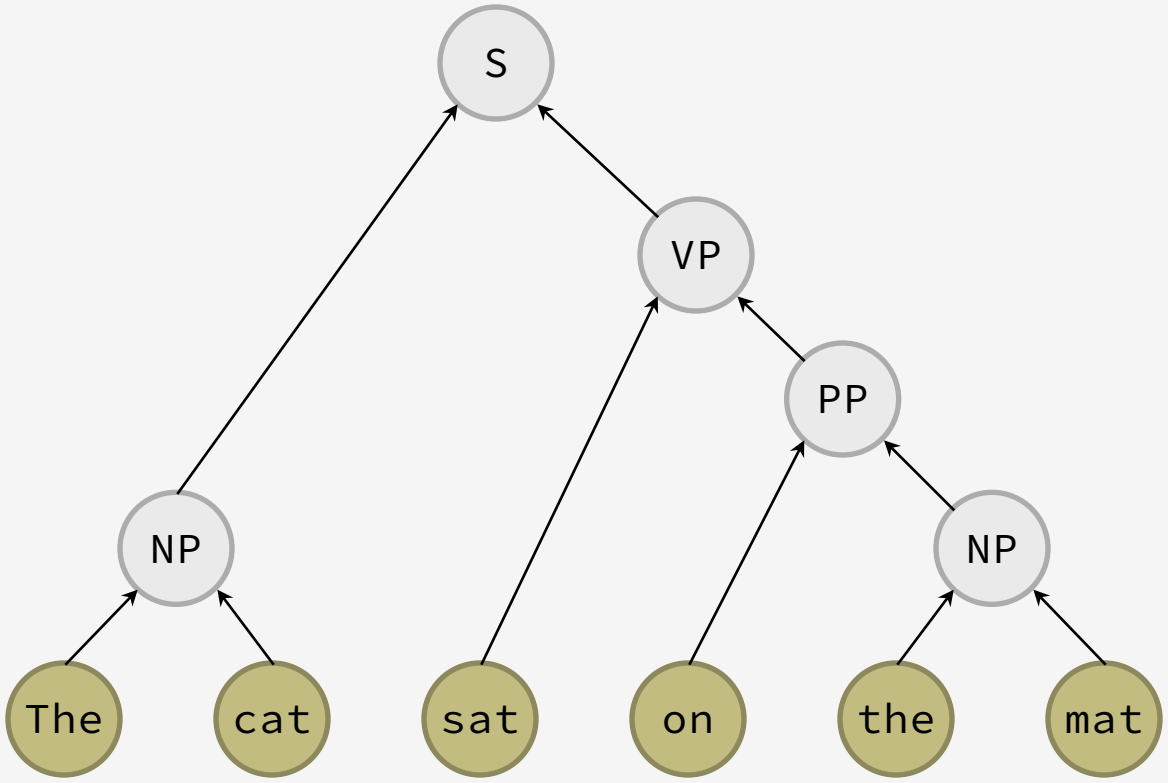
- Common in natural language processing

- Variable sentence length

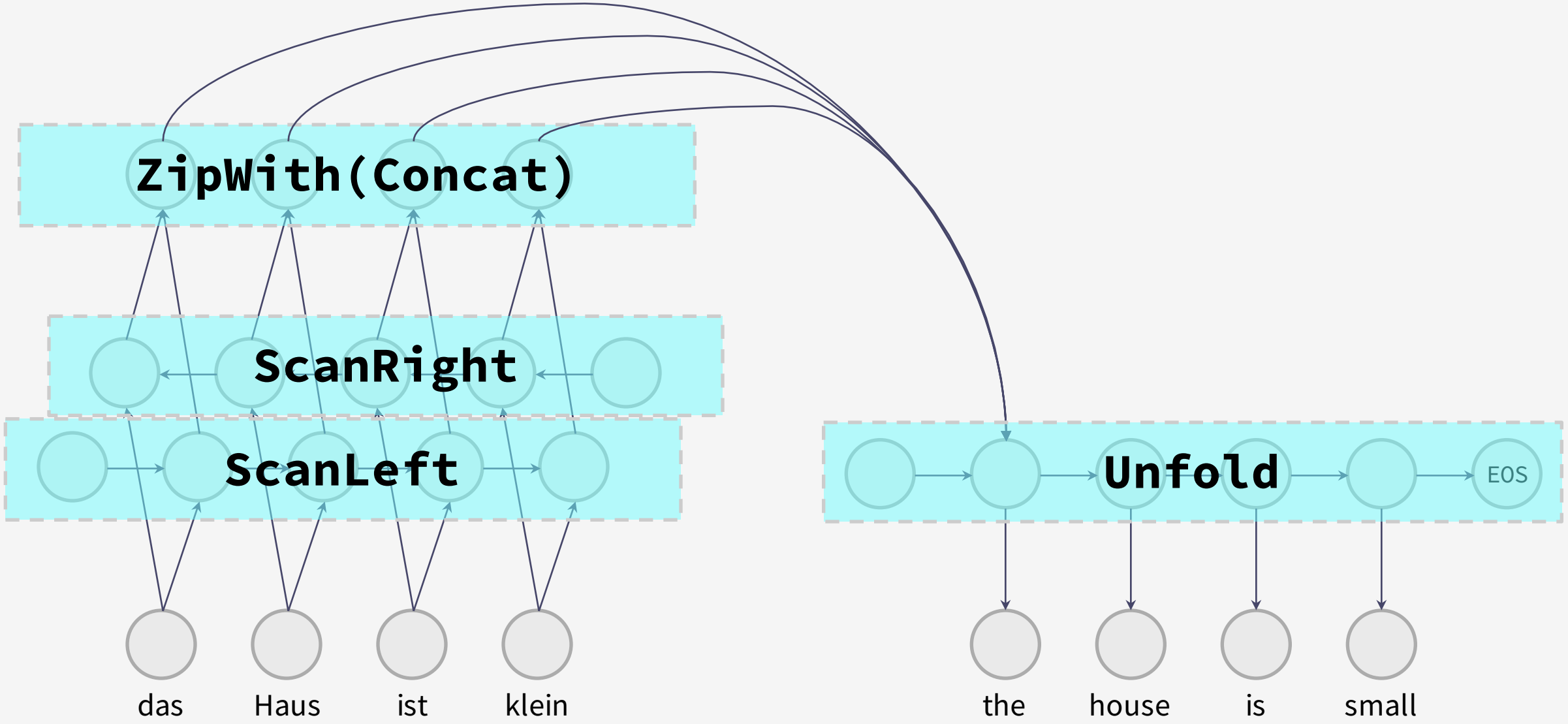


Neural networks with dynamic structures

- Distinct syntactic structures

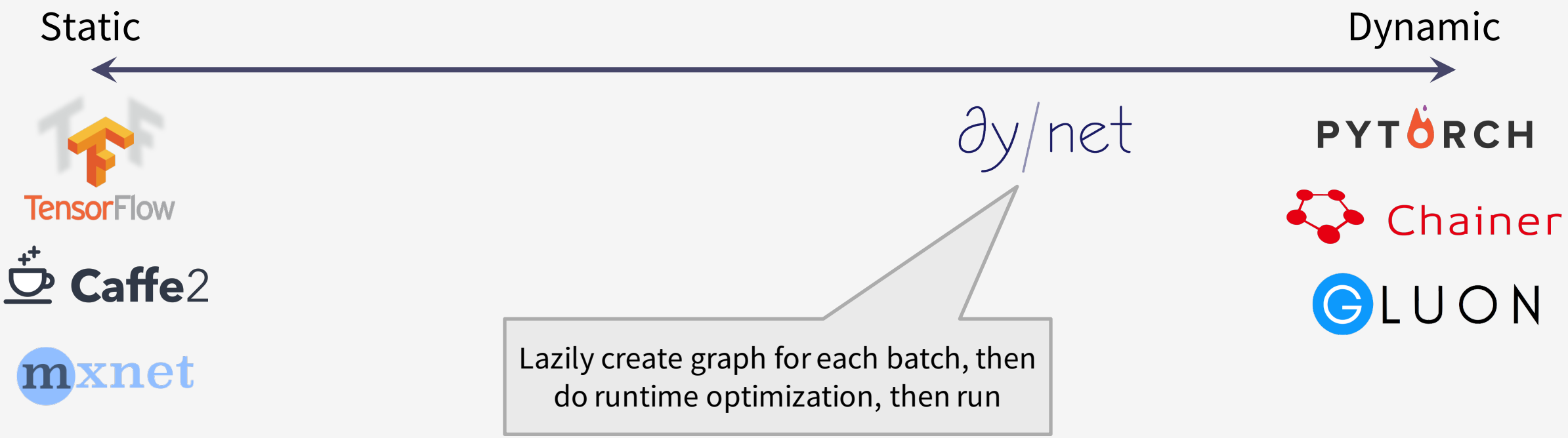


Example: Neural machine translation (Seq2Seq)



Static vs dynamic computation graphs

- Static: Construct graph once, interpret later
  - Difficult to implement dynamic neural networks
- Dynamic: Compute as you construct the graph
  - Lost the ability to do runtime optimization



User control of evaluation

```
sealed trait Expr[X] {
  /**
   * Gets the value of this expression given an implicit
   computation instance, while forcing this expression to be
   evaluated strictly in that specific computation instance.
   */
  def value(implicit comp: Expr ~> Id): X = comp(this)
}
```

Normally the computation graph is constructed lazily. Once `value` is called, the interpreter is forced to compute to graph to this node.

User control of evaluation

```
val ŷ = x |> Layer1 |> Sigmoid |> Layer2 |> Softmax
val loss = (y, ŷ) |> CrossEntropy
```

Constructs the computation graph (Declaratively, no actually computation executed)

```
given (x := xValue, y := yValue) { implicit computation =>

  val lossValue = loss.value
  averageLoss += lossValue
  ……

}
```

Calling value in implicit computation scope forces the interpreter to evaluate

Future work

- Towards a fully-fledged Scala deep learning engine
  - Automatic batching (fusion of computation graphs)
  - Complete GPU support
  - Garbage collection (off-heap memory & GPU memory)
  - Distributed learning (through Spark?)

- Help needed!

Q & A