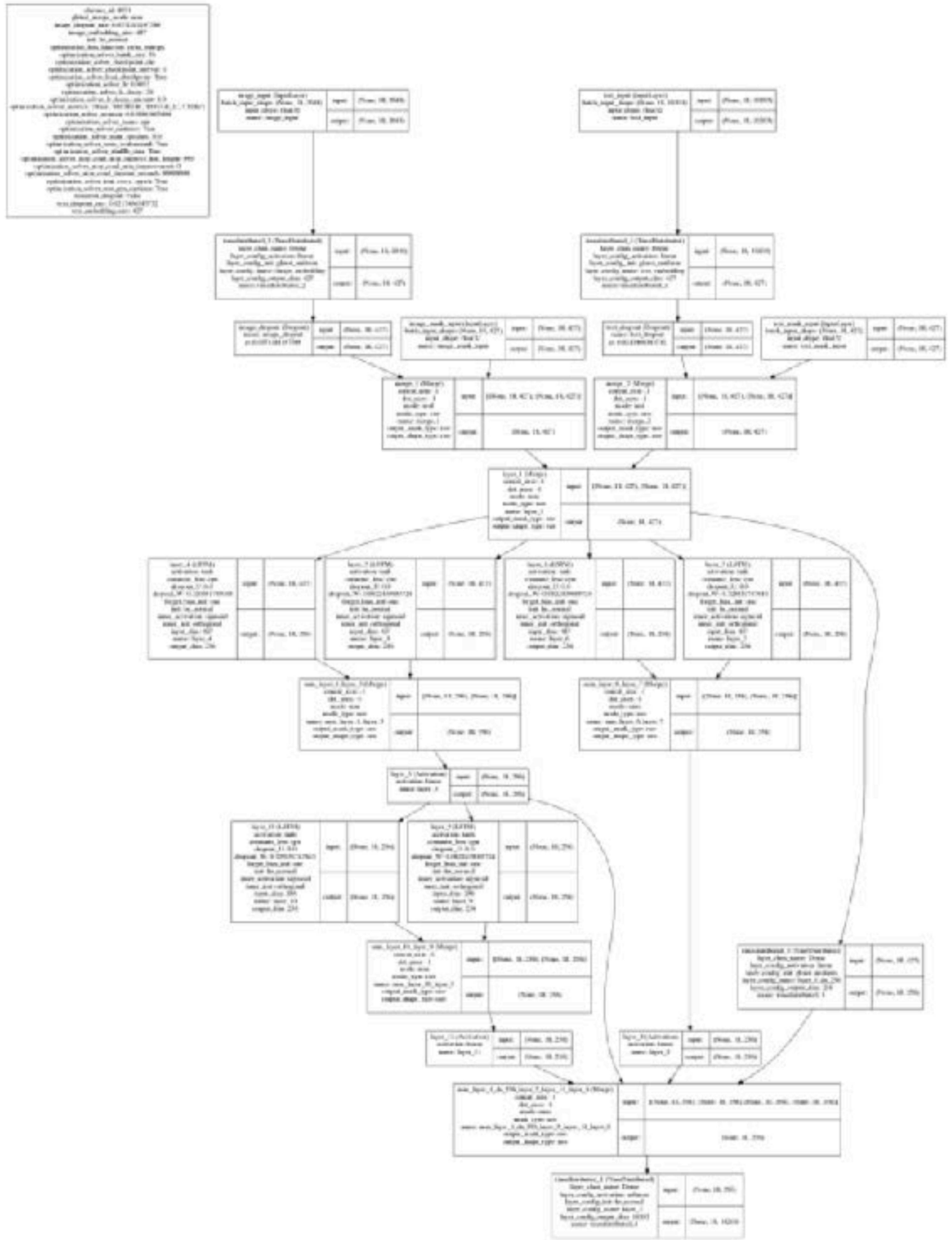

ThoughtWorks®
TALK IS CHEAP
SHOW ME THE CODE
begin with a story

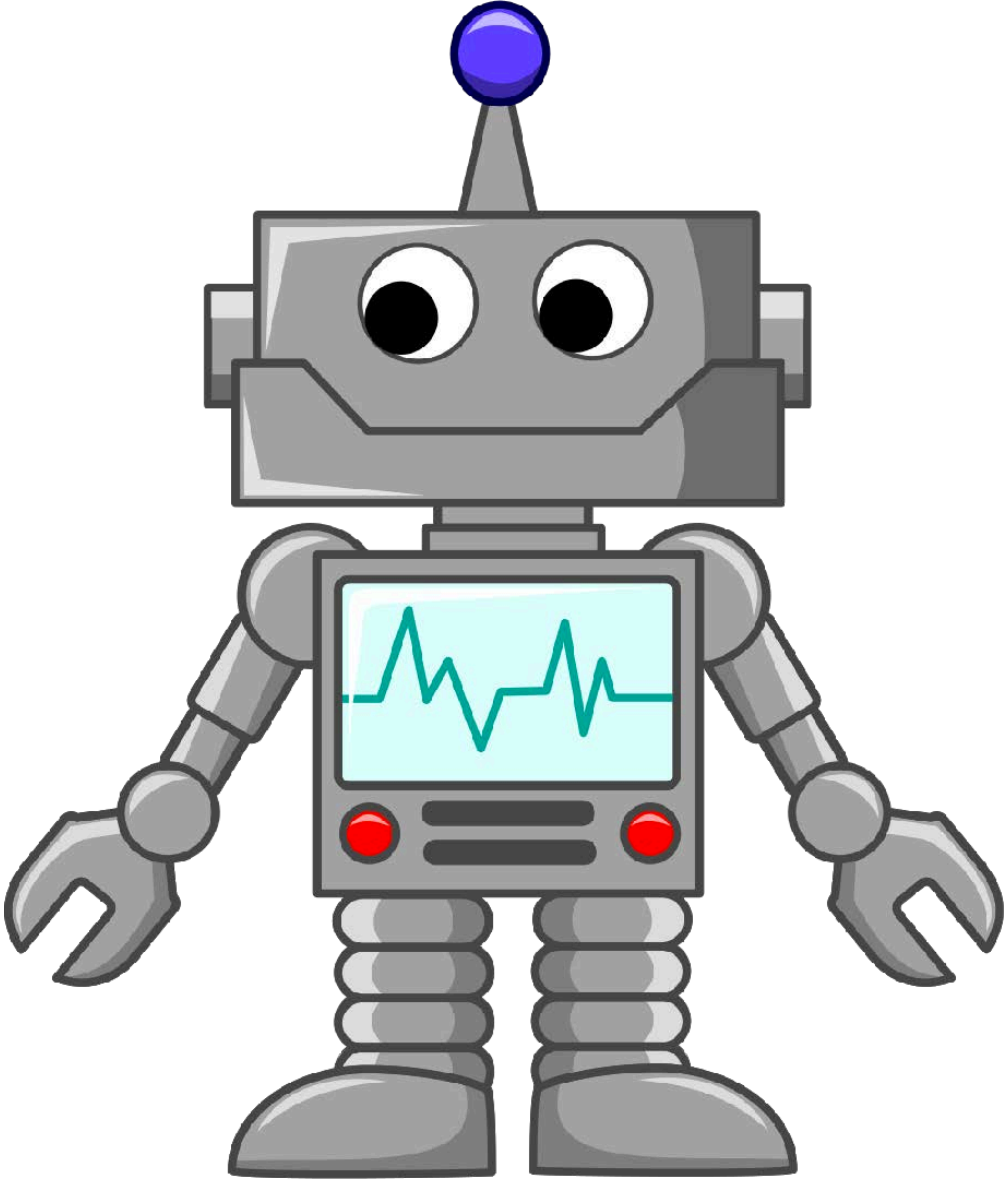
GROWTH OF DATA SCIENCE

Node Hyperparameter	Range
Number of Filters	[32, 256]
Dropout Rate	[0, 0.7]
Initial Weight Scaling	[0, 2.0]
Kernel Size	{1, 3}
Max Pooling	{True, False}
Global Hyperparameter	**Range**
Learning Rate	[0.0001, 0.1]
Momentum	[0.68, 0.99]
Hue Shift	[0, 45]
Saturation/Value Shift	[0, 0.5]
Saturation/Value Scale	[0, 0.5]
Cropped Image Size	[26, 32]
Spatial Scaling	[0, 0.3]
Random Horizontal Flips	{True, False}
Variance Normalization	{True, False}
Nesterov Accelerated Gradient	{True, False}

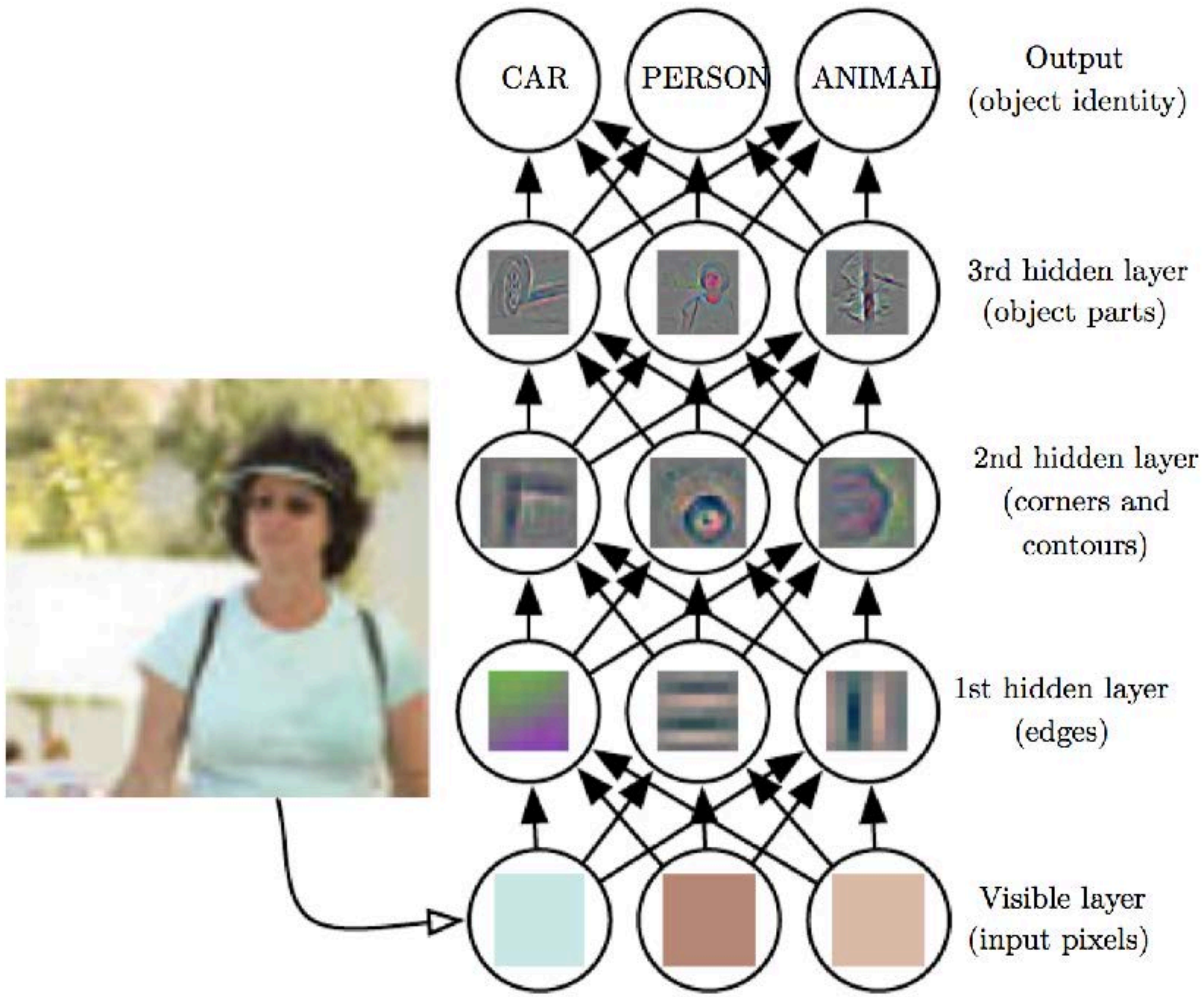
Table 1: Node and global hyperparameters evolved in the CIFAR-10 domain.



https://arxiv.org/pdf/1703.00548v1.pdf



DEEP LEARNING



Deep Learning

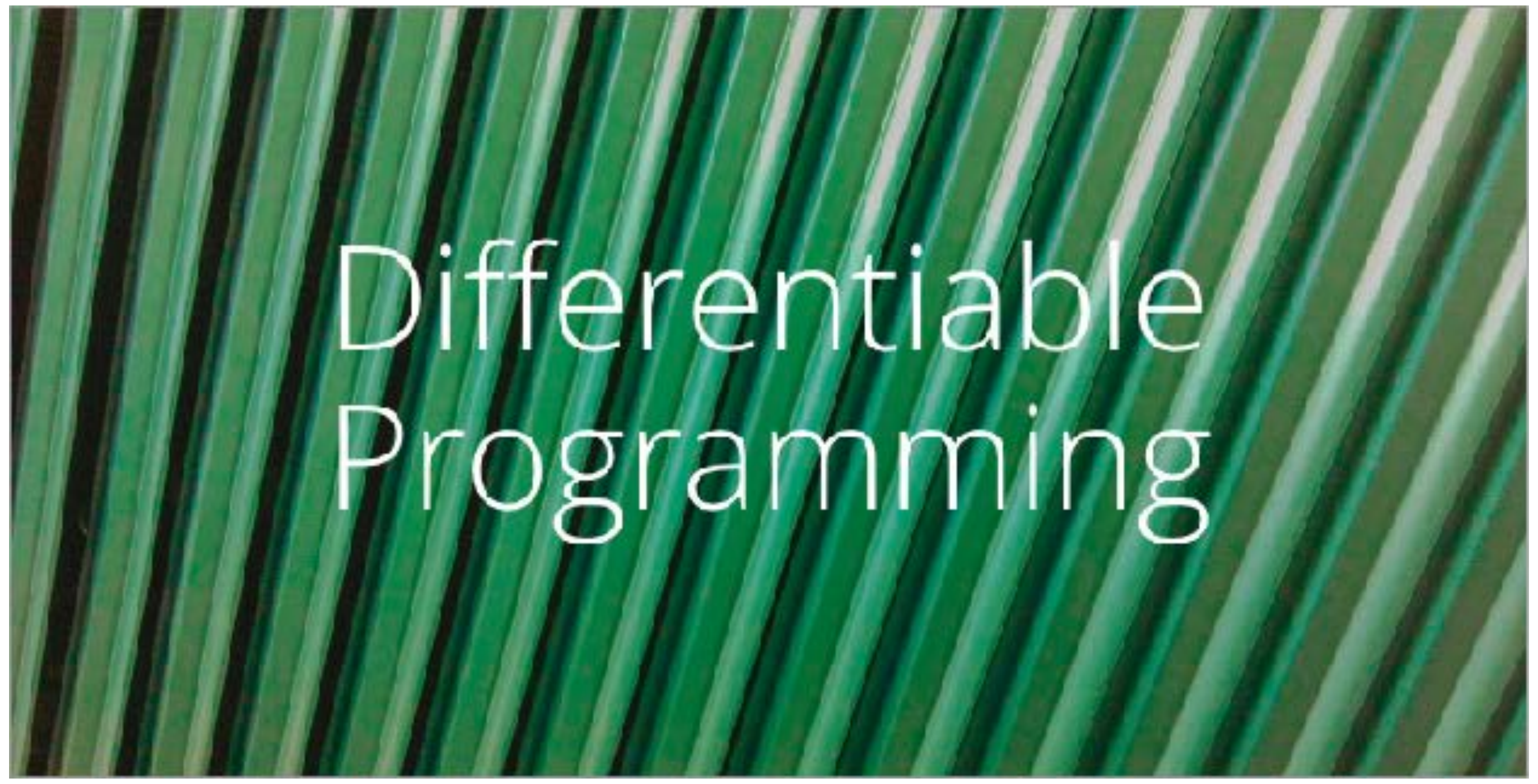
Ian Goodfellow Yoshua Bengio Aaron Courville

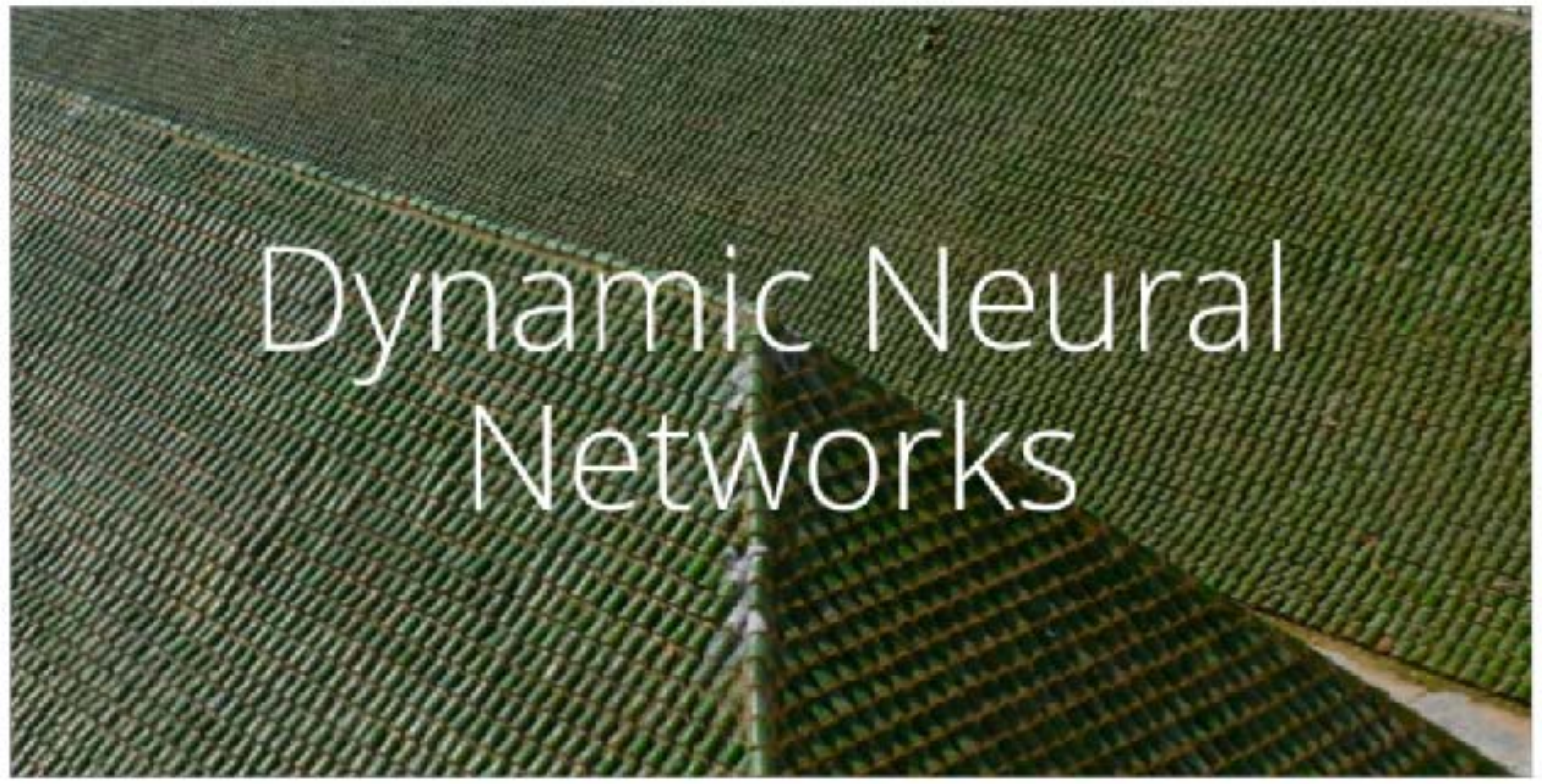
ThoughtWorks®

NEURO NETWORK IS FUNCTIONAL PROGRAMMING

deeplearning.scala

DeepLearning.scala





IQ TEST ROBOT

What is the next number in the sequence?

3,4,5,?

13,19,25,?

THE SIGNATURE OF IQ TEST ROBOT

```
def guessNextNumber(question: Seq[Double]): DoubleLayer = ???


println(guessNextNumber(Seq(3, 4, 5)).predict.blockingAwait)

println(guessNextNumber(Seq(13, 19, 25)).predict.blockingAwait)
```

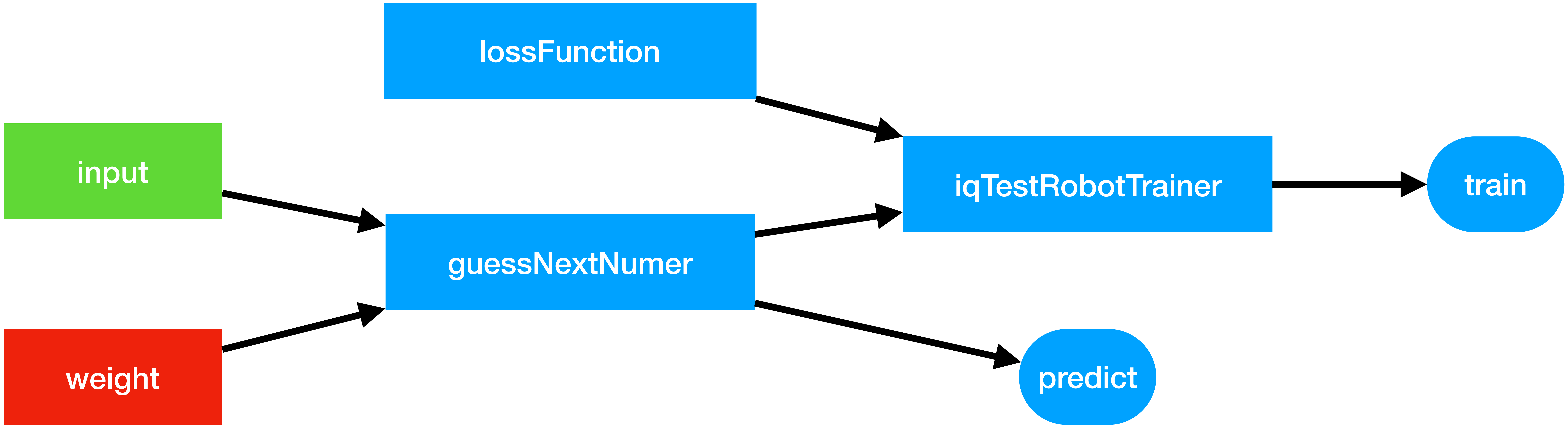
TRAINING THE IQ TEST ROBOT

```
def lossFunction(robotAnswer: DoubleLayer, expectedAnswer: Double): DoubleLayer = ???

def iqTestRobotTrainer(question: Seq[Double], expectedAnswer: Double): DoubleLayer = {
  val robotAnswer = guessNextNumber(question)
  lossFunction(robotAnswer, expectedAnswer)
}

iqTestRobotTrainer(Seq(3, 4, 5), 6).train.blockingAwait
iqTestRobotTrainer(Seq(13, 19, 25), 31).train.blockingAwait
```

TRAINING IN TENSORFLOW

```
pred_x = model(X, w_h, w_o)
loss = tf.reduce_mean(tf.nn.lossfunction(pred_x, Y))
train_op = tf.train.GradientDescentOptimizer(0.05).minimize(loss)
```

FUNCTIONS ARE COMPOSED



TRAINING THE IQ TEST ROBOT

```scala
def lossFunction(robotAnswer: DoubleLayer, expectedAnswer: Double): DoubleLayer = ???

def iqTestRobotTrainer(question: Seq[Double], expectedAnswer: Double): DoubleLayer = {
  val robotAnswer = guessNextNumber(question)
  lossFunction(robotAnswer, expectedAnswer)
}

iqTestRobotTrainer(Seq(3, 4, 5), 6).train.blockingAwait
iqTestRobotTrainer(Seq(13, 19, 25), 31).train.blockingAwait
```

THE STRUCTURE OF THE IQ TEST ROBOT

```scala
// A vector of weights that are lazily initialized
val weights: Stream[DoubleWeight] =
Stream.continually(DoubleWeight(math.random))

// The predictor implemented with map/reduce
def guessNextNumber(question: Seq[Double]): DoubleLayer = {
  (question zip weights).map {
    case (element, weight) => element * weight
  }.reduce(_ + _)
}

// Square loss function
def lossFunction(robotAnswer: DoubleLayer, expectedAnswer:
Double): DoubleLayer = {
  val difference: DoubleLayer = robotAnswer - expectedAnswer
  difference * difference
}
```

SOLUTION IN TENSORFLOW

```
def model(X, w_h, w_o):
    h = tf.nn.relu(tf.matmul(X, w_h))
    return tf.matmul(h, w_o)
```

THE STRUCTURE OF THE IQ TEST ROBOT

```scala
// A vector of weights that are lazily initialized
val weights: Stream[DoubleWeight] =
Stream.continually(DoubleWeight(math.random))

// The predictor implemented with map/reduce
def guessNextNumber(question: Seq[Double]): DoubleLayer = {
  (question zip weights).map {
    case (element, weight) => element * weight
  }.reduce(_ + _)
}

// Square loss function
def lossFunction(robotAnswer: DoubleLayer, expectedAnswer:
Double): DoubleLayer = {
  val difference: DoubleLayer = robotAnswer - expectedAnswer
  difference * difference
}
```

META-PROGRAMMING

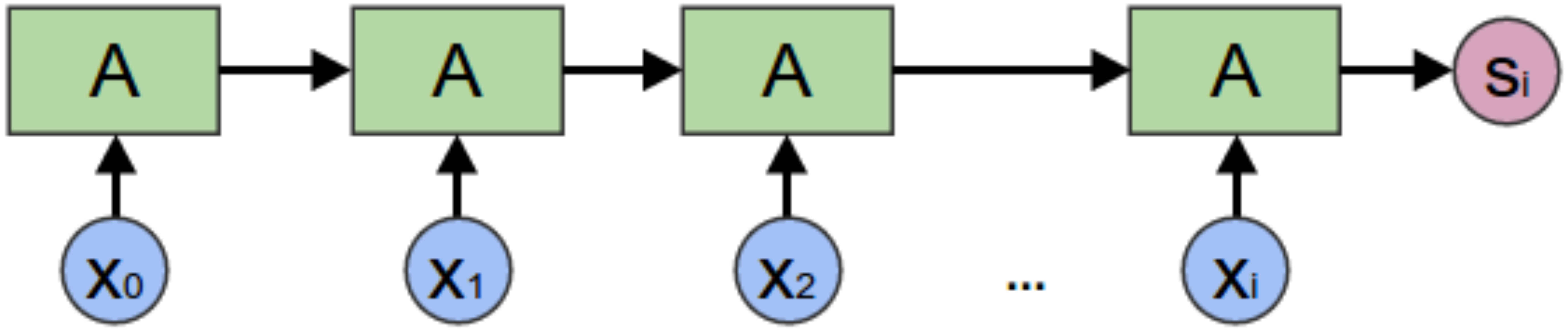
https://prateekvjoshi.files.wordpress.com/2014/04/1-main.png

ThoughtWorks®

CORRESPONDING BETWEEN NEURAL NETWORK AND FUNCTIONAL PROGRAMMING

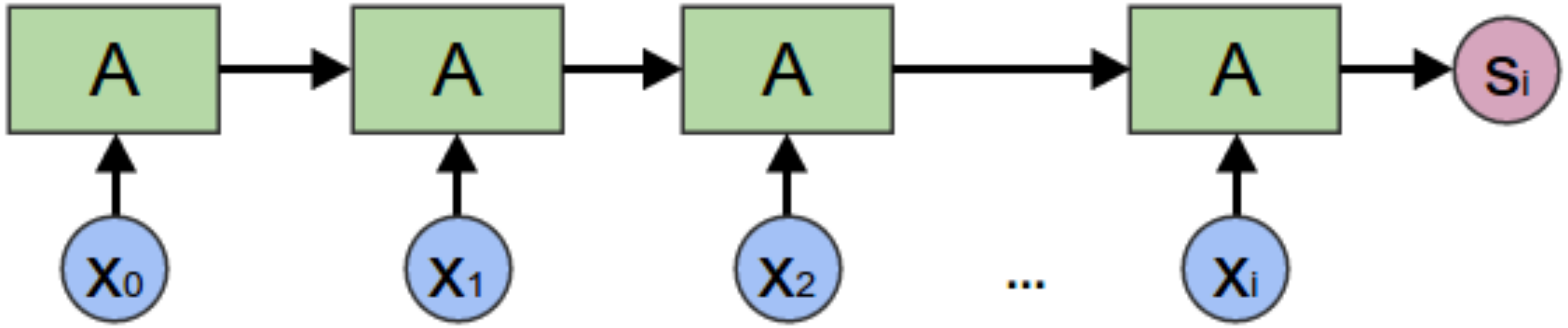
yes it is

ENCODING RECURRENT NEURAL NETWORKS



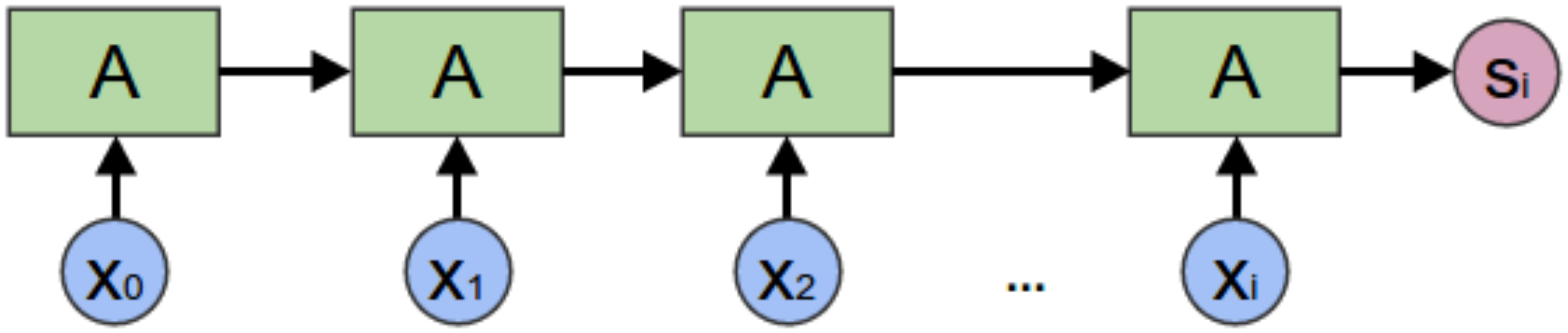
https://colah.github.io/posts/2015-09-NN-Types-FP/

ENCODING RECURRENT NEURAL NETWORKS



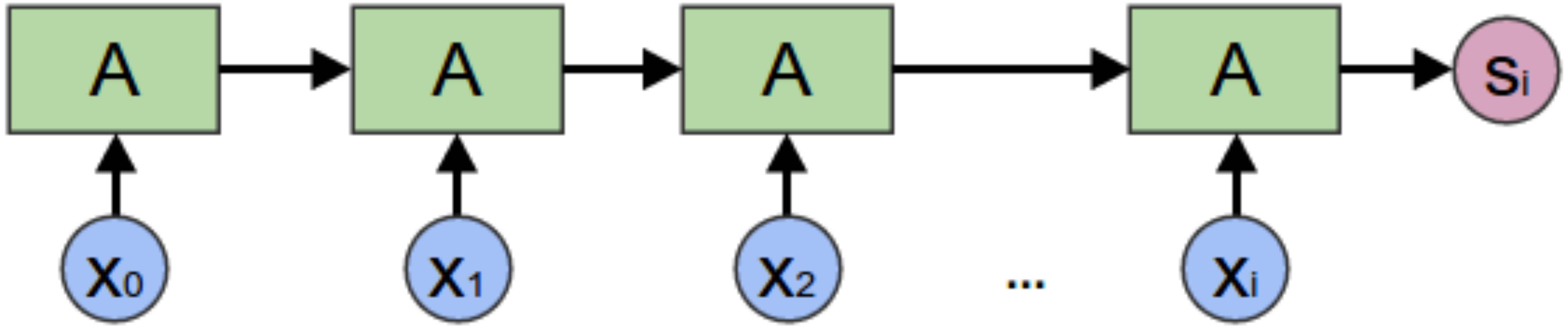

```scala
package scala.collection
trait Seq[A] {
  def foldLeft[B](z: B)(step: (B, A) => B): B
}
```

ENCODING RECURRENT NEURAL NETWORKS



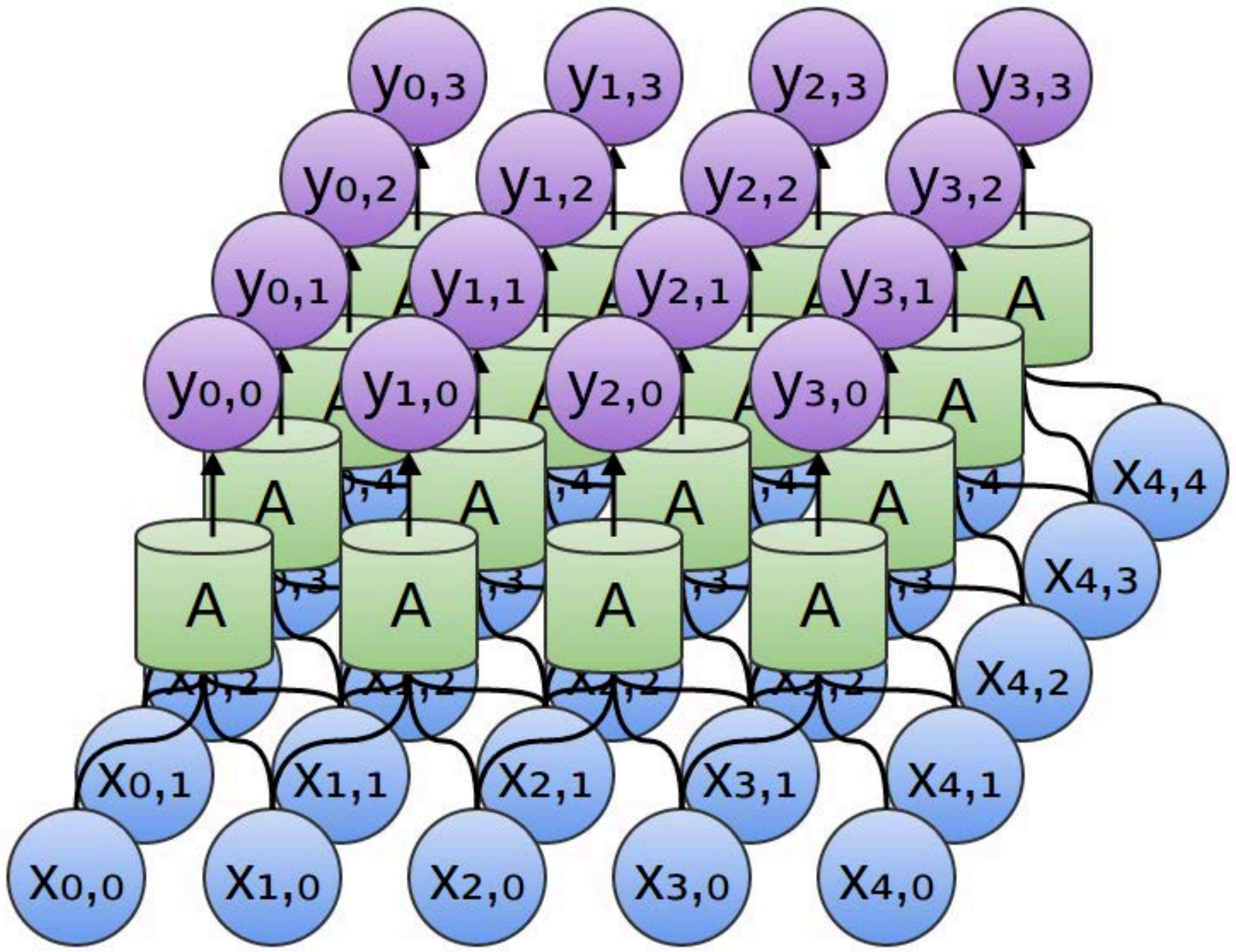

```scala
type Input = Seq[DoubleLayer]
type State = Seq[DoubleLayer]

def step(hiddenState: State, xi: Input): State = ???

def encodingRNN(x: Seq[Input]): State = {
  val initialState: State = Seq.empty[DoubleLayer]
  x.foldLeft(initialState)(step)
}
```

ENCODING RECURRENT NEURAL NETWORKS



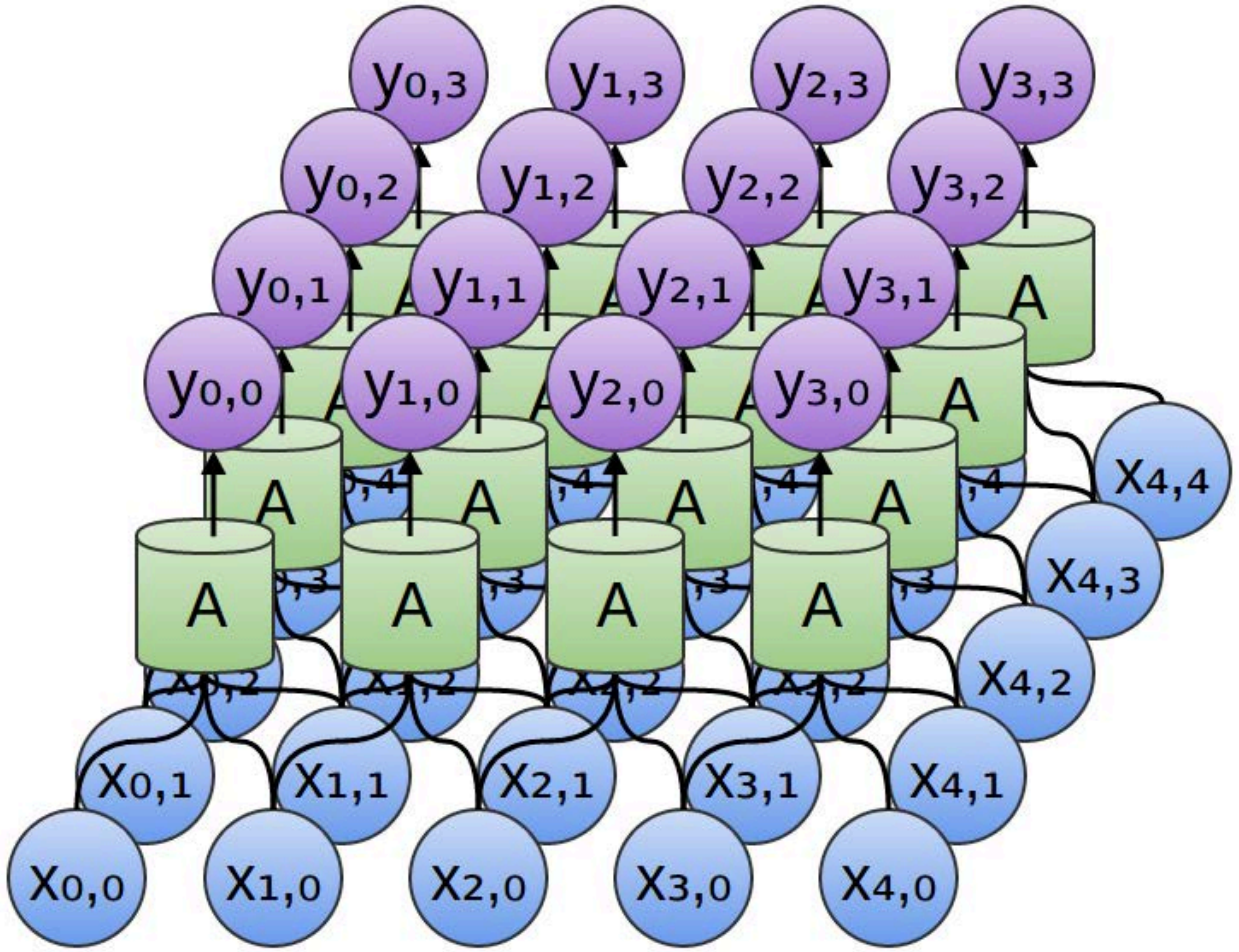

```scala
val NumberOfOutputFeatures = 10

val weight: Seq[Seq[DoubleWeight]] = Seq.fill(NumberOfOutputFeatures)(
  Stream.continually(DoubleWeight(math.random))
)

def step(hiddenState: State, xi: Input): State = {
  tanh(matrixMultiply(hiddenState ++ xi, weight))
}
```

TWO DIMENSIONAL CONVOLUTIONAL NETWORK



https://colah.github.io/posts/2015-09-NN-Types-FP/

TWO DIMENSIONAL CONVOLUTIONAL NETWORK




```
type Input = Seq[DoubleLayer]
type Output = Seq[DoubleLayer]

def kernel2x2(xi: Input, xj: Input, xk: Input, xl:
Input): Output

def cnn2d(x: List[List[Input]]): List[Output]) = {
  val x00 = x
  val x01 = x.map(_.tail)
  val x10 = x.tail
  val x11 = x.tail.map(_.tail)
  (x zip x01 zip x10 zip x11).map {
    case (((xi, xj), xk), xl) => kernel2x2(xi, xj, xk,
xl)
  }
}
```

https://colah.github.io/posts/2015-09-NN-Types-FP/

NEURO NETWORK IS FUNCTIONAL PROGRAMMING

Deep Learning Name	Functional Name
Learned Vector	Holes in metaprogramming
Encoding RNN	Fold
Generating RNN	Unfold
General RNN	Accumulating Map
Bidirectional RNN	Zipped Left/Right Accumulating Maps
Conv Layer	“Window Map”
TreeNet	scanr

ThoughtWorks®

MONADIC DEEP LEARNING

dynamically

MIXTURE OF EXPERTS

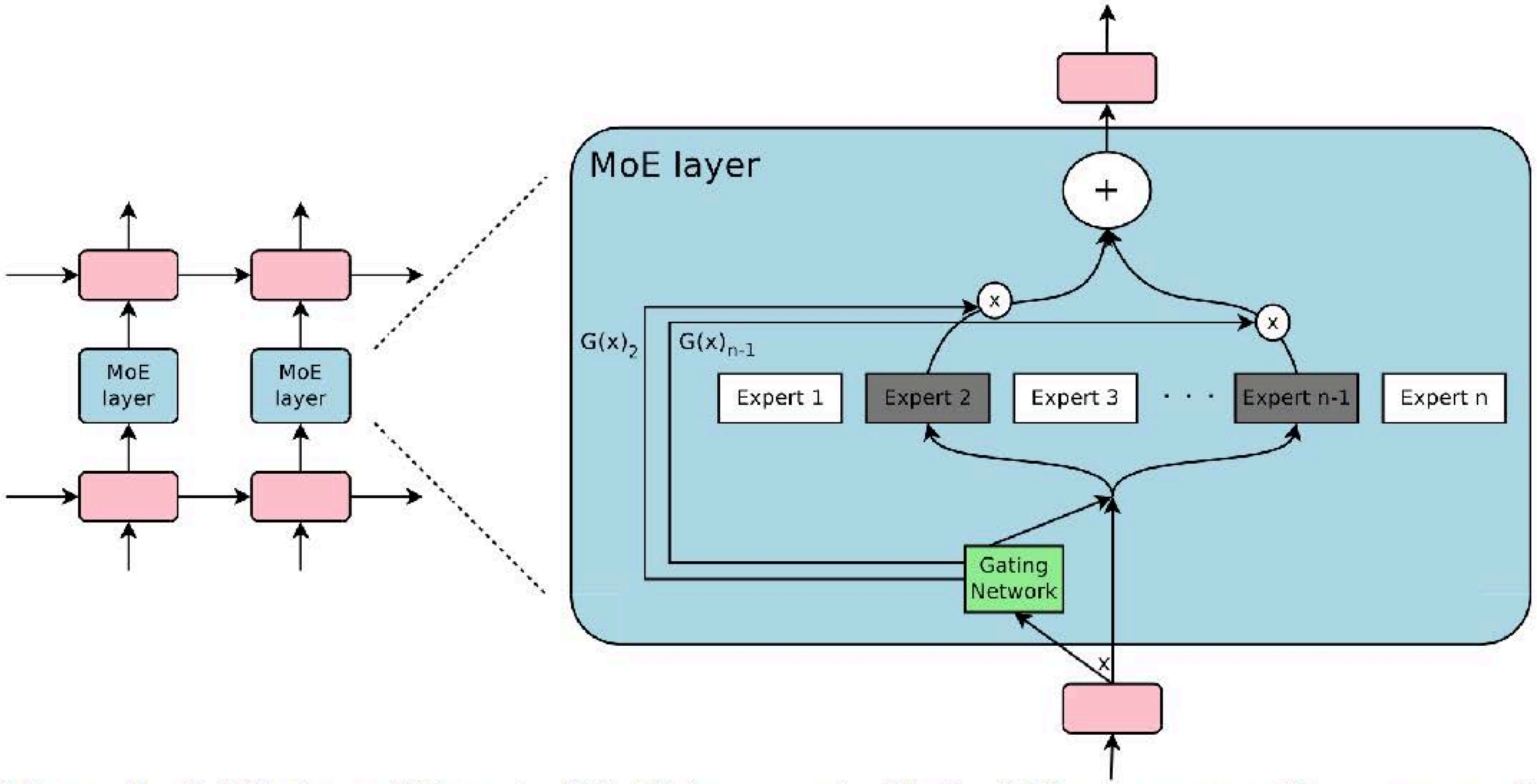


Figure 1: A Mixture of Experts (MoE) layer embedded within a recurrent language model. In this case, the sparse gating function selects two experts to perform computations. Their outputs are modulated by the outputs of the gating network.

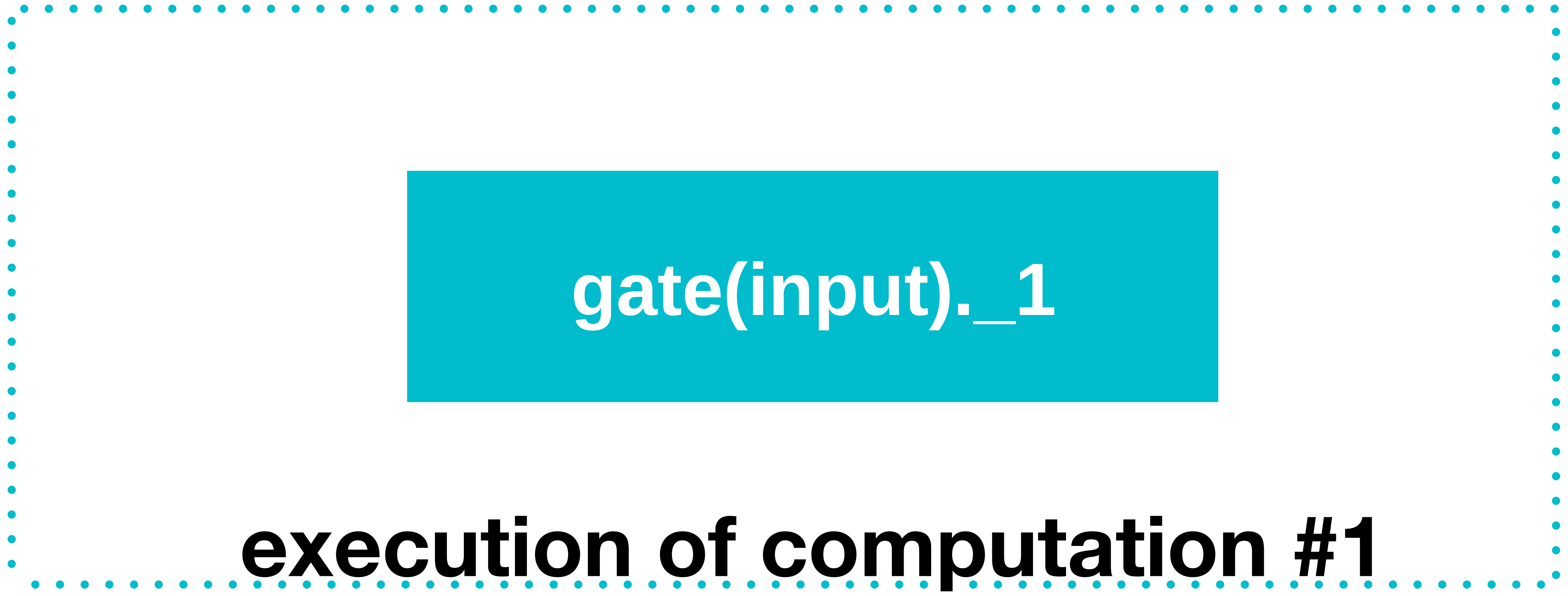
https://arxiv.org/pdf/1701.06538.pdf

NAIVE GATED NET

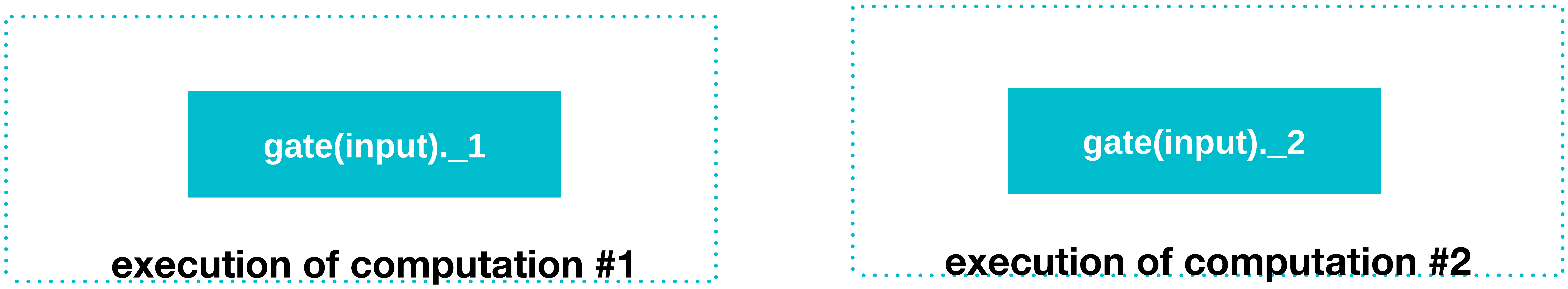
```
type Input = INDArrayLayer
type Output = INDArrayLayer
type Condition = (DoubleLayer, DoubleLayer)

def leftSubnet(input: Input): Output = ???
def rightSubnet(input: Input): Output = ???
def gate(input: Input): Condition = ???

def naiveGatedNet(input: Input): Output = {
  val condition = gate(input)
  if (condition._1.predict.blockingAwait > condition._2.predict.blockingAwait) {
    condition._1 * leftSubnet(input)
  } else {
    condition._2 * rightSubnet(input)
  }
}
```

NAIVE GATED NET

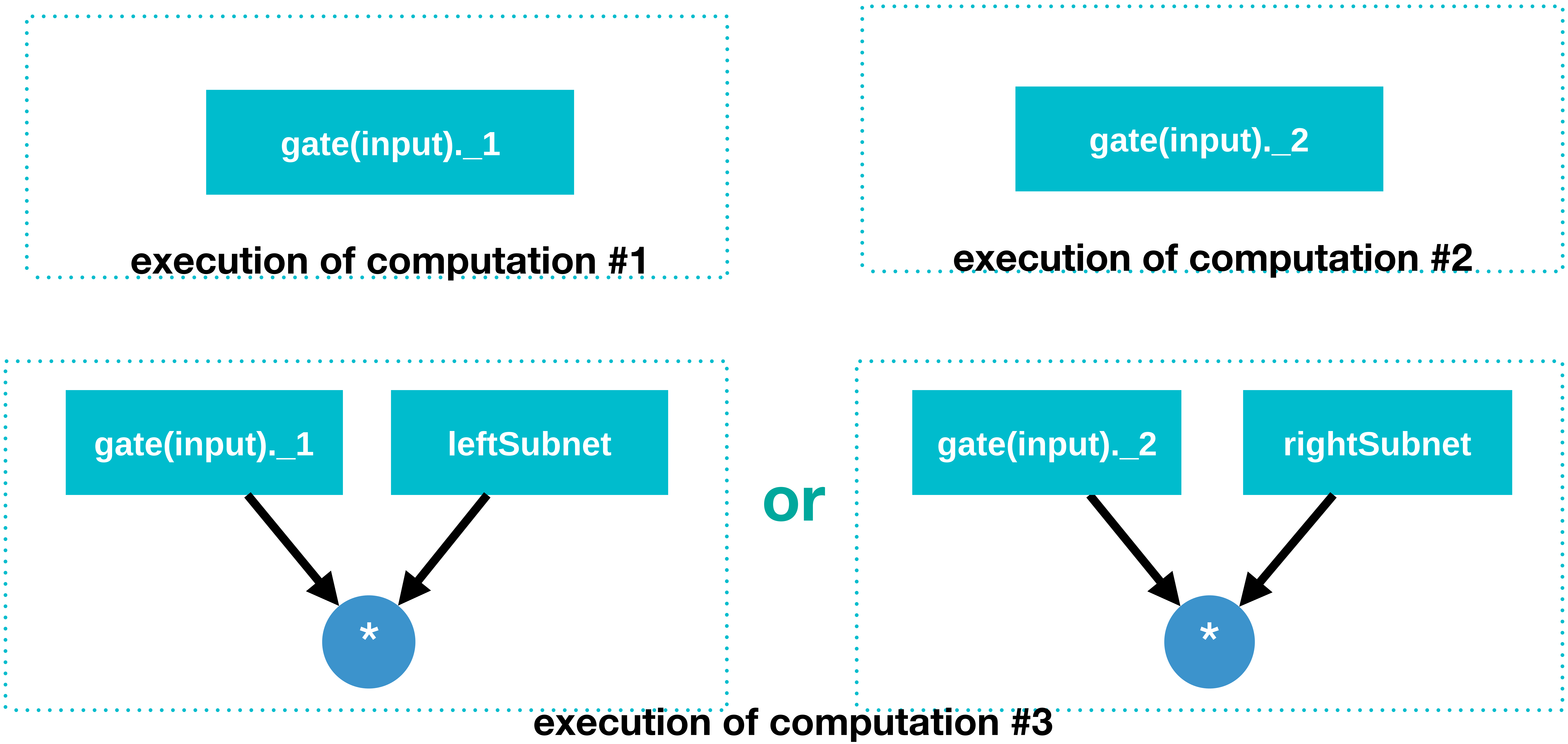
NAIVE GATED NET



NAIVE GATED NET



NAIVE GATED NET



MONAD

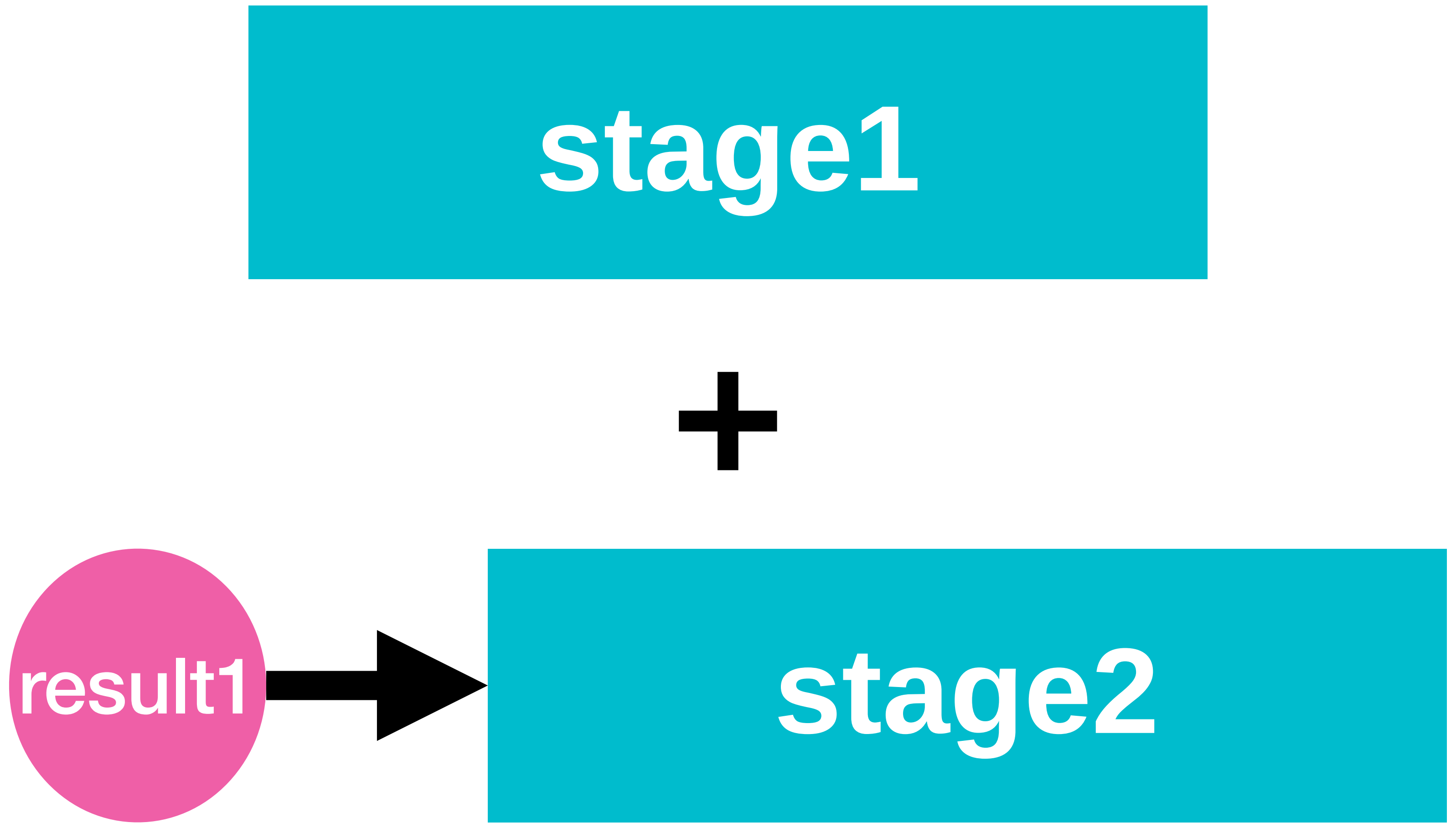
```
class Do[A] {
  def flatMap[B](f: A => Do[B]): Do[B]
}
```

THE USAGE OF FLATMAP

```
val stage1: Do[Stage1Result] = ???
def stage2(result1: Stage1Result): Do[Stage2Result] = ???

val flatMapped: Do[Stage2Result] = stage1.flatMap(stage2)
```

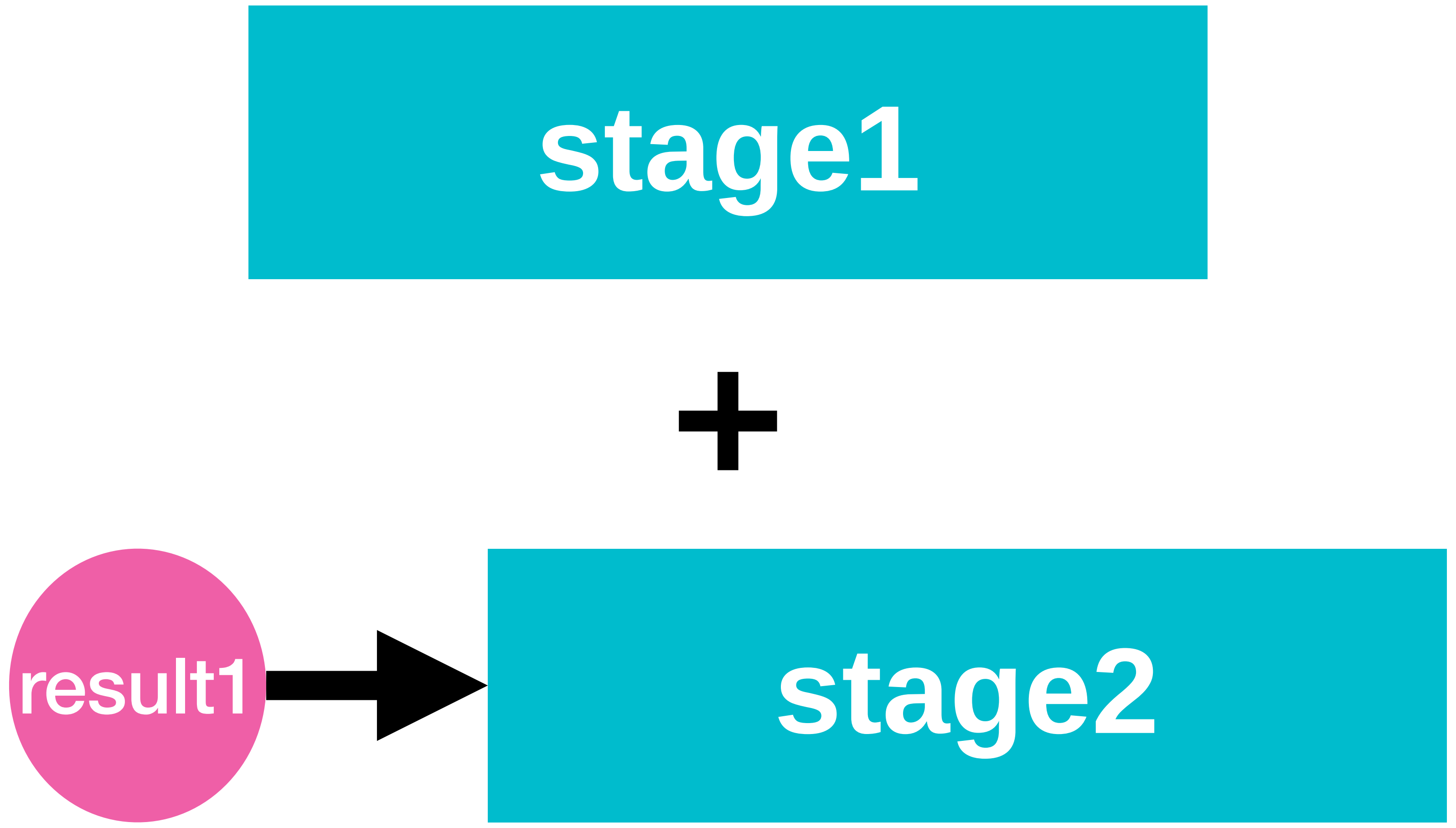
THE USAGE OF FLATMAP

```
val stage1: Do[Stage1Result] = ???
def stage2(result1: Stage1Result): Do[Stage2Result] = ???

val flatMapped: Do[Stage2Result] = stage1.flatMap(stage2)
```



THE USAGE OF FLATMAP

```scala
val stage1: Do[Stage1Result] = ???
def stage2(result1: Stage1Result): Do[Stage2Result] = ???

val flatMapped: Do[Stage2Result] = stage1.flatMap(stage2)
```



flatMap

THE USAGE OF FLATMAP

```
val stage1: Do[Stage1Result] = ???
def stage2(result1: Stage1Result): Do[Stage2Result] = ???

val flatMapped: Do[Stage2Result] = stage1.flatMap(stage2)
```

MONADIC GATED NET

```scala
type Stage1Result = (Tape[Double, Double], Tape[Double, Double])
def monadicGatedNet(input: Input): Output = {
  val condition = gate(input)
  val stage1: Do[Stage1Result] = (condition._1.forward) tuple2 (condition._2.forward)
  def stage2(pair: Stage1Result) = {
    if (pair._1.data > pair._2.data) {
      (condition._1 * leftSubnet(input)).forward
    } else {
      (condition._2 * rightSubnet(input)).forward
    }
  }

  val gatedForward = stage1.flatMap(stage2)
  INDArrayLayer(gatedForward)
}
```

MONADIC GATED NET

```
class Do[A] {
  def tuple2[B](fb: Do[B]): Do[(A, B)]
}
```

MONADIC GATED NET

```
type Stage1Result = (Tape[Double, Double], Tape[Double, Double])
def monadicGatedNet(input: Input): Output = {
  val condition = gate(input)
  val stage1: Do[Stage1Result] = (condition._1.forward) tuple2 (condition._2.forward)
  def stage2(pair: Stage1Result) = {
    if (pair._1.data > pair._2.data) {
      (condition._1 * leftSubnet(input)).forward
    } else {
      (condition._2 * rightSubnet(input)).forward
    }
  }

  val gatedForward = stage1.flatMap(stage2)
  INDArrayLayer(gatedForward)
}
```

PARALLEL GATED NET

```
val stage1 = (condition._1.forward) tuple2 (condition._2.forward)
```

PARALLEL GATED NET

```
val st  e1 = (condition  1.forward) tuple2 (conditio  _2.forward)
```

PARALLEL GATED NET

```
val stage1 = (condition._1.forward) tuple2 (condition._2.forward)
```

```
val Parallel(stage1) = Parallel(condition._1.forward) tuple2 Parallel(condition._2.forward)
```

PARALLEL GATED NET

```scala
type Stage1Result = (Tape[Double, Double], Tape[Double, Double])
def monadicGatedNet(input: Input): Output = {
  val condition = gate(input)
  val Parallel(stage1) = Parallel(condition._1.forward) tuple2 Parallel(condition._2.forward)

  def stage2(pair: Stage1Result) = {
    if (pair._1.data > pair._2.data) {
      (condition._1 * leftSubnet(input)).forward
    } else {
      (condition._2 * rightSubnet(input)).forward
    }
  }

  val gatedForward = stage1.flatMap(stage2)
  INDArrayLayer(gatedForward)
}
```