Δ ▽

:-dyna.

Tim Vieira, Matthew Francis-Landau,
Nathaniel Wesley Filardo, Farzad Khorasani,
and Jason Eisner
Dyna:
Toward a Self-Optimizing Declarative Language for Machine Learning Applications
Dyna:
Toward a Self-Optimizing Declarative Language for Machine Learning Applications
Dyna:
Toward a Self-Optimizing Declarative Language for Machine Learning Applications
Dyna:
Toward a Self-Optimizing Declarative Language
for Machine Learning Applications

Faster to implement
Dyna:
Toward a Self-Optimizing Declarative Language for Machine Learning Applications

Faster execution

ML Δ \n \n:-dyna. PL

Faster to implement
Dyna:
Toward a Self-Optimizing Declarative Language for Machine Learning Applications

Faster execution

ML  \(\Delta\ \nabla\)  PL

:--dyna.

Faster to implement
Outline
Outline

• Why Declarative Programming?
Outline

• Why Declarative Programming?
• Quick introduction to the Dyna language
Outline

• Why Declarative Programming?
• Quick introduction to the Dyna language
• Automatic optimization of Dyna programs
Declarative Programming
Declarative Programming

A programming paradigm where the programmer specifies **what** to compute and leaves **how** to compute it to a solver.
Declarative Programming

A programming paradigm where the programmer specifies what to compute and leaves how to compute it to a solver.

- Examples: SQL, Prolog/Datalog, Mathematica, Regex, TensorFlow/Theano
Declarative Programming

A programming paradigm where the programmer specifies what to compute and leaves how to compute it to a solver.

- Examples: SQL, Prolog/Datalog, Mathematica, Regex, TensorFlow/Theano
- Solver seeks an efficient strategy (e.g., SQL query planning)
Why declarative programming?
Why declarative programming?

- Many ML algorithms have a concise declarative program
Why declarative programming?

• Many ML algorithms have a concise declarative program
• There are many choices to make when writing a fast program
Why declarative programming?

- Many ML algorithms have a concise declarative program
- There are many choices to make when writing a fast program
  - Loop orders
Why declarative programming?

• Many ML algorithms have a concise declarative program
• There are many choices to make when writing a fast program
  • Loop orders
  • Data structures (e.g., hash map, dense array, linked list)
Why declarative programming?

• Many ML algorithms have a concise declarative program
• There are many choices to make when writing a fast program
  • Loop orders
  • Data structures (e.g., hash map, dense array, linked list)
  • Global execution strategy (e.g., depth vs. breadth-first search)
Why declarative programming?

• Many ML algorithms have a concise declarative program
• There are many choices to make when writing a fast program
  • Loop orders
  • Data structures (e.g., hash map, dense array, linked list)
  • Global execution strategy (e.g., depth vs. breadth-first search)
  • Parallelization opportunities
Why declarative programming?

• Many ML algorithms have a concise declarative program
• There are many choices to make when writing a fast program
  • Loop orders
  • Data structures (e.g., hash map, dense array, linked list)
  • Global execution strategy (e.g., depth vs. breadth-first search)
  • Parallelization opportunities
• Manually experimenting with all possibilities is time consuming
Why declarative programming?

• Many ML algorithms have a concise declarative program
• There are many choices to make when writing a fast program
  • Loop orders
  • Data structures (e.g., hash map, dense array, linked list)
  • Global execution strategy (e.g., depth vs. breadth-first search)
  • Parallelization opportunities
• Manually experimenting with all possibilities is time consuming
  • Programmers usually only implement one
Why declarative programming?

• Many ML algorithms have a concise declarative program
• There are many choices to make when writing a fast program
  • Loop orders
  • Data structures (e.g., hash map, dense array, linked list)
  • Global execution strategy (e.g., depth vs. breadth-first search)
  • Parallelization opportunities
• Manually experimenting with all possibilities is time consuming
  • Programmers usually only implement one
• Researchers don’t have time to optimize the efficiency of their code
  • We can do better with automatic optimization
Why not optimize Python/Java/C++ etc.?
Why not optimize Python/Java/C++ etc.?  

• Less flexibility  
  • Choices of loop orders / data structures already decided by the human programmer
Why not optimize Python/Java/C++ etc.?

• Less flexibility
  • Choices of loop orders / data structures already decided by the human programmer

• Semantics of the program are not invariant to
  • Changing execution and loop order
    • Eager vs. lazy evaluation, top-down vs bottom-up evaluation.
Why not optimize Python/Java/C++ etc.?

• Less flexibility
  • Choices of loop orders / data structures already decided by the human programmer

• Semantics of the program are not invariant to
  • Changing execution and loop order
    • Eager vs. lazy evaluation, top-down vs bottom-up evaluation.
  • Data structures
Why not optimize Python/Java/C++ etc.?

• Less flexibility
  • Choices of loop orders / data structures already decided by the human programmer

• Semantics of the program are not invariant to
  • Changing execution and loop order
    • Eager vs. lazy evaluation, top-down vs bottom-up evaluation.
  • Data structures
  • Concurrency
Why not optimize Python/Java/C++ etc.?

• Less flexibility
  • Choices of loop orders / data structures already decided by the human programmer

• Semantics of the program are not invariant to
  • Changing execution and loop order
    • Eager vs. lazy evaluation, top-down vs bottom-up evaluation.
  • Data structures
  • Concurrency

• Difficult to reliably discover long range interactions in a program
What is Dyna?
What is Dyna?

• Declarative language
What is Dyna?

• Declarative language
• Based on weighted logic programming
What is Dyna?

• Declarative language
• Based on weighted logic programming
• Prolog / Datalog like syntax
  • Uses pattern matching to define computation graphs
What is Dyna?

• Declarative language
• Based on weighted logic programming
• Prolog / Datalog like syntax
  • Uses pattern matching to define computation graphs
• Reactive
What is Dyna?

• Declarative language
• Based on weighted logic programming
• Prolog / Datalog like syntax
  • Uses pattern matching to define computation graphs
• Reactive
• Dyna programs are close to their mathematical description
  • Similar to functional programs
Dyna Day 1
Dyna Day 1

\[ a = b \times c. \]

a will be kept up to date if b or c changes. (Reactive)
Dyna Day 1

\[ a = b \times c. \]

a will be kept up to date if b or c changes. (Reactive)

\[ b += x. \]

\[ b += y. \quad \text{equivalent to} \quad b = x+y. \quad \text{(almost)} \]

b is a sum of two variables. Also kept up to date.
Dyna Day 1

\[ a = b \times c. \]

\( a \) will be kept up to date if \( b \) or \( c \) changes. (Reactive)

\[
\text{b} \; += \; \text{x}.
\]

\[
\text{b} \; += \; \text{y}. \quad \text{equivalent to} \quad \text{b} = \text{x} + \text{y}. \quad \text{(almost)}
\]

\( b \) is a sum of two variables. Also kept up to date.

\[
\text{c} \; += \; \text{z}(1) .
\]

\[
\text{c} \; += \; \text{z}(2) .
\]

\[
\text{c} \; += \; \text{z}(3) .
\]
Dyna Day 1

\[ a = b \times c. \]

*a* will be kept up to date if *b* or *c* changes. \((\text{Reactive})\)

\[ b += x. \]

\[ b += y. \quad \text{equivalent to} \quad b = x+y. \quad \text{\(\text{almost}\)} \]

*b* is a sum of *two* variables. Also kept up to date.

\[ c += z(1). \]
\[ c += z(2). \]
\[ c += z(3). \]

*c* is a sum of *all* defined \(z(\ldots)\) values.

\[ a \text{ "patterns" the capitalized N matches anything} \]
Dyna Day 1

\[ a = b \times c. \]

\( a \) will be kept up to date if \( b \) or \( c \) changes. (Reactive)

\[ b += x. \]
\[ b += y. \quad \text{equivalent to } b = x+y. \quad \text{(almost)} \]

\( b \) is a sum of two variables. Also kept up to date.

\[ c += z(1). \]
\[ c += z(2). \]
\[ c += z(3). \]
\[ c += z("four"). \]
\[ c += z(\text{foo(bar,5)}). \]

\( c \) is a sum of all defined \( z(\ldots) \) values.
More interesting use of “patterns”
More interesting use of “patterns”

\[ a(I) = b(I) \ast c(I). \]

• pointwise multiplication
More interesting use of “patterns”

\[ a(I) = b(I) \ast c(I). \]

- pointwise multiplication

\[ a += b(I) \ast c(I). \]

- dot product; could be sparse

\[ a = \sum_i b_i \ast c_i \]
More interesting use of “patterns”

\[ a(I) = b(I) \times c(I). \]
- pointwise multiplication

\[ a \textbf{+=} b(I) \times c(I) . \]
- dot product; could be sparse

\[
\left( a = \sum_i b_i \times c_i \right)
\]
More interesting use of “patterns”

\[ a(I) = b(I) \times c(I). \]
- pointwise multiplication

\[ a += b(I) \times c(I). \]
- dot product; could be sparse

\[ a(I,K) += b(I,J) \times c(J,K). \]
- matrix multiplication; could be sparse
  - \( J \) is free on the right-hand side, so we sum over it
More interesting use of “patterns”

\[ a(I) = b(I) \ast c(I). \]

- pointwise multiplication

\[ a += b(I) \ast c(I). \]

- dot product; could be sparse

\[ a(I,K) += b(I,J) \ast c(J,K). \]

- matrix multiplication; could be sparse

- \( J \) is free on the right-hand side, so we sum over it

\[ a = \sum_i b_i \ast c_i \]

\[ a_{i,k} = \sum_j b_{i,j} \ast c_{j,k} \]
More interesting use of “patterns”

\[ a(I) = b(I) \times c(I). \]
- pointwise multiplication

\[ a += b(I) \times c(I). \]
- dot product; could be sparse

\[ a(I,K) += b(I,J) \times c(J,K). \]
- matrix multiplication; could be sparse
- \( J \) is free on the right-hand side, so we sum over it
Dyna vs. Prolog
Dyna vs. Prolog

Prolog has Horn clauses:

\[
a(I,K) \leftarrow b(I,J) , c(J,K).
\]
Dyna vs. Prolog

Prolog has Horn clauses:

\[ a(I, K) :- b(I, J) , c(J, K). \]

Dyna has "Horn equations":

\[ a(I, K) += b(I, J) * c(J, K). \]
Dyna vs. Prolog

Prolog has Horn clauses:

\[ a(I,K) :- b(I,J), c(J,K). \]

Dyna has “Horn equations”:

\[ a(I,K) += b(I,J) * c(J,K). \]
Dyna vs. Prolog

Prolog has Horn clauses:

\[ a(I,K) :- b(I,J), c(J,K). \]

Dyna has “Horn equations”:

\[ a(I,K) += b(I,J) * c(J,K). \]

prove a value for it
e.g., a real number,
but could be any term
Dyna vs. Prolog

Prolog has Horn clauses:

\[ a(I,K) :\text{=} b(I,J) , c(J,K) . \]

Dyna has “Horn equations”:

\[ a(I,K) += b(I,J) \times c(J,K) . \]

prove a \textbf{value} for it
e.g., a real number,
but could be any term
definition from other values
\[ b \times c \] only has value when \[ b \] and \[ c \] do
if no values enter into +=, then \( a \) gets no value
Dyna vs. Prolog

Prolog has Horn clauses:
\[ a(I,K) :- b(I,J), c(J,K). \]

Dyna has “Horn equations”:
\[ a(I,K) + = b(I,J) \ast c(J,K). \]

prove a value for it
e.g., a real number,
but could be any term
definition from other values
b\ast c only has value when b and c do
if no values enter into + = , then a gets no value

Like Prolog:
Allow nested terms
Syntactic sugar for lists, etc.
Turing-complete
Dyna vs. Prolog

Prolog has Horn clauses:

\[ a(I,K) :- b(I,J), c(J,K). \]

Dyna has “Horn equations”:

\[ a(I,K) += b(I,J) \times c(J,K). \]

prove a value for it
e.g., a real number,
but could be any term

definition from other values
\( b \times c \) only has value when \( b \) and \( c \) do
if no values enter into \( += \), then \( a \) gets no value

Like Prolog:
Allow nested terms
Syntactic sugar for lists, etc.
Turing-complete

Unlike Prolog:
Terms can have values
Terms are evaluated in place
Not just backtracking!
Shortest path
Shortest path

distance(X) min= edge(X, Y) + distance(Y).
distance(start) min= 0.
path_length := distance(end).
Shortest path

distance(X) \text{ min} = \text{edge}(X, Y) + \text{distance}(Y).
distance(\text{start}) \text{ min} = 0.
\text{path\_length} := \text{distance}(\text{end}).

distance("a", "b") = 10.
distance("b", "c") = 2.
distance("c", "d") = 7.
distance("d", "b") = 1.
start = "a".
end = "d".
Shortest path

distance(X) min= edge(X, Y) + distance(Y).
distance(start) min= 0.
path_length := distance(end).

\[
\text{distance}(x) = \min_{y \in \text{edges}(x, \cdot)} \text{edge}(x, y) + \text{distance}(y)
\]

\[
\text{distance(Start)} = 0
\]

Path length = distance(End)
Shortest path

distance(X) \text{ min } = \text{ edge}(X, Y) + \text{ distance}(Y).
distance(\text{start}) \text{ min } = 0.
\text{path_length} := \text{ distance}(\text{end}).

Variables not present in the head of an expression are aggregated over like with the dot product example.
Shortest path

distance(X) \ \text{min} = \text{edge}(X, Y) + \text{distance}(Y).
distance(\text{start}) \ \text{min} = 0
\text{path\_length} = : = \text{distance}(\text{end}).

distance(x) = \min_{y \in \text{edges}(x, \cdot)} \text{edge}(x, y)
distance(\text{Start}) = 0
\text{Path\ length} = \text{distance}(\text{End})

Here the "\text{min} = " aggregator only keeps the minimal value that we have computed
Shortest path

\[ \text{min} = \text{edge}(X, Y) + \text{distance}(Y). \]
\[ \text{min} = 0. \]
\[ \text{path length} := \text{distance}(\text{end}). \]

\[
\text{distance}(x) = \min_{y \in \text{edges}(x, \cdot)} \text{edge}(x, y) + \text{distance}(y)
\]

\[ \text{distance(Start)} = 0 \]
\[ \text{Path length} = \text{distance(End)} \]

Note: Aggregation was already present in our mathematical definition.
Shortest path

distance(X)  \text{min} = \text{edge}(X, Y) + \text{distance}(Y).
distance(start) \text{min} = 0.
\text{path}\_\text{length} \quad ::= \text{distance}(end).

After this converges we can \textit{query} the state of the Dyna program.
Shortest path

\[
\text{distance}(X) \quad \text{min} = \text{edge}(X, Y) + \text{distance}(Y).
\]
\[
\text{distance}(\text{start}) \quad \text{min} = 0.
\]
\[
\text{path}\_\text{length} \quad ::= \text{distance}(\text{end}).
\]

After this converges we can \textit{query} the state of the Dyna program.

? \text{path}\_\text{length}

Length at the end
Shortest path

distance(X) \text{ min} = \text{edge}(X, Y) + \text{distance}(Y).
distance(\text{start}) \text{ min} = 0.
\text{path\_length} := \text{distance}(\text{end}).

After this converges we can \textit{query} the state of the Dyna program.

? \text{path\_length}
? \text{distance("c")}

The distance of some other vertex
Shortest path

\[
\text{distance}(X) \quad \text{min} = \text{edge}(X, Y) + \text{distance}(Y).
\]
\[
\text{distance}(\text{start}) \quad \text{min} = 0.
\]
\[
\text{path}_\text{length} \quad := \text{distance}(\text{end}).
\]

After this converges we can \textit{query} the state of the Dyna program.

? \text{path}_\text{length}
? \text{distance}("c")
? \text{distance}(X)

All of the vertices
Shortest path

distance(X) \ min= \ edge(X, Y) + distance(Y).
distance(start) \ min= \ 0.
path_length \ := \ distance(end).

After this converges we can \textit{query} the state of the Dyna program.

? path_length
? distance("c")
? distance(X)
? distance(X) > 7

All the vertices more than 7 away
Shortest path

distance(\(X\)) \text{ min} = \text{edge}(\(X\), \(Y\)) + \text{distance}(\(Y\)).
distance(\text{start}) \text{ min} = 0.
path\_length := \text{distance}(\text{end}).

After this converges we can \textit{query} the state of the Dyna program.

? path\_length
? distance("c")
? distance(\(X\))
? distance(\(X\)) > 7
? \text{edge}("a", \(X\))

All of the edges leaving "a"
Aggregators
Aggregators

• Associative/commutative:
  • b += a(X). % number
  • c max= a(X).
  • q |= p(X). % boolean
  • r &= p(X).
Aggregators

• Associative/commutative:
  • $b \ += \ a(X)$. % number
  • $c \ \text{max}= \ a(X)$.
  • $q \ |= \ p(X)$. % boolean
  • $r \ \&= \ p(X)$.

• Require uniqueness:
  • $d \ = \ b+c$. 
Aggregators

• Associative/commutative:
  • \( b += a(X) \). % number
  • \( c \max= a(X) \).
  • \( q \|= p(X) \). % boolean
  • \( r \&= p(X) \).

• Require uniqueness:
  • \( d = b+c \).

• Last one wins:
  • \( \text{fly}(X) := \text{true if bird}(X) \).
  • \( \text{fly}(X) := \text{false if penguin}(X) \).
  • \( \text{fly}(\text{bigbird}) := \text{false} \).
Aggregators

• Associative/commutative:
  • $b += a(X)$. % number
  • $c \text{ max}= a(X)$.
  • $q \text{ |= } p(X)$. % boolean
  • $r \text{ &= } p(X)$.

• Require uniqueness:
  • $d = b+c$.

• Last one wins:
  • $\text{fly}(X) := \text{true}$ if $\text{bird}(X)$.
  • $\text{fly}(X) := \text{false}$ if $\text{penguin}(X)$.
  • $\text{fly}(\text{bigbird}) := \text{false}$.

• Choose any value:
  • $e ?= b$.
  • $e ?= c$. 

• Choose any value:
Aggregators

• Associative/commutative:
  • \( b += a(X) \). % number
  • \( c \max= a(X) \).
  • \( q |= p(X) \). % boolean
  • \( r \&= p(X) \).

• Require uniqueness:
  • \( d = b+c \).

• Last one wins:
  • \( \text{fly}(X) := \text{true if } \text{bird}(X) \).
  • \( \text{fly}(X) := \text{false if } \text{penguin}(X) \).
  • \( \text{fly}(	ext{bigbird}) := \text{false} \).

• Choose any value:
  • \( e \?= b \).
  • \( e \?= c \).

• User definable aggregators
  • \( a(X) \textbf{smiles}= b(X, Z) \).
Aggregators

- Associative/commutative:
  - $b := a(X)$. % number
  - $c \max= a(X)$.
  - $q \mid= p(X)$. % boolean
  - $r \&= p(X)$.

- Require uniqueness:
  - $d = b+c$.

- Last one wins:
  - $fly(X) := \text{true if bird}(X)$.
  - $fly(X) := \text{false if penguin}(X)$.
  - $fly(bigbird) := \text{false}$.

- Choose any value:
  - $e ?= b$.
  - $e ?= c$.

- User definable aggregators
  - $a(X) \text{smiles}= b(X, Z)$.
  - (Just define all of the operation of an commutative semigroup)
Neural Convolutional Layer

Input image

Convolution output

1 1 1 0 0
0 1 1 1 0
0 0 1 1 1
0 1 1 0 0
0 0 1 1 0

http://deeplearning.stanford.edu/wiki/index.php/Feature_extraction_using_convolution
Neural Convolutional Layer

\[ h_{i,j} = \sigma \left( \sum_{m\in[-1,1],n\in[-1,1]} i_{m+i,n+j} \ast w_{m,n} \right) \]

Input image

Convolution output

http://deeplearning.stanford.edu/wiki/index.php/Feature_extraction_using_convolution
Neural Convolutional Layer

\[ h_{i,j} = \sigma \left( \sum_{m \in [-1,1], n \in [-1,1]} i_{m+i,n+j} \ast w_{m,n} \right) \]

Input image

Output

http://deeplearning.stanford.edu/wiki/index.php/Feature_extraction_using_convolution
Neural Convolutional Layer

\[ h_{i,j} = \sigma \left( \sum_{m \in [-1,1], n \in [-1,1]} i_{m+i,n+j} \cdot w_{m,n} \right) \]

Input image:

<table>
<thead>
<tr>
<th>1 x_1</th>
<th>1 x_0</th>
<th>1 x_1</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 x_0</td>
<td>1 x_1</td>
<td>1 x_0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0 x_1</td>
<td>0 x_0</td>
<td>1 x_1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Learned feature weights

Output:

4
Neural Convolutional Layer

\[ h_{i,j} = \sigma \left( \sum_{m \in [-1,1], n \in [-1,1]} i_{m+i,n+j} \ast w_{m,n} \right) \]

Some nonlinearity

http://deeplearning.stanford.edu/wiki/index.php/Feature_extraction_using_convolution
Neural Convolutional Layer

$$h_{i,j} = \sigma \left( \sum_{m \in [-1,1], n \in [-1,1]} i_{m+i,n+j} \ast w_{m,n} \right)$$

Input image

Convolution output

Convolution output

http://deeplearning.stanford.edu/wiki/index.php/Feature_extraction_using_convolution
Neural Convolutional layer

\[
h_{i,j} = \sigma \left( \sum_{m \in [-1,1], n \in [-1,1]} i_{m+i, n+j} \ast w_{m,n} \right)
\]

(activation(I, J)).

(X) := 1 / (1 + \exp(-X)).
Neural Convolutional layer

\[ h_{i,j} = \sigma \left( \sum_{m\in[-1,1],n\in[-1,1]} i_{m+i,n+j} \ast w_{m,n} \right) \]

\[ \sigma(X) := 1 / (1 + \exp(-X)). \]

= \sigma(\text{activation}(I, J)).

\text{activation}(I, J) += \text{input}(I + M, J + N) \ast \text{weight}(M, N).

\text{weight}(DX,DY) := \text{random}(*,-1,1) \text{ for } DX:-1..1, \text{ DY:-1..1.}
Neural Convolutional layer

\[ h_{i,j} = \sigma \left( \sum_{m \in [-1,1], n \in [-1,1]} i_{m+i,n+j} \cdot w_{m,n} \right) \]

\[ \sigma(X) := 1 / (1 + \exp(-X)) \]

= \sigma(\text{activation}(I, J))

\text{activation}(I, J) += \text{input}(I + M, J + N) \cdot \text{weight}(M, N).

\text{weight}(DX, DY) := \text{random}(\ast, -1, 1) \text{ for } DX:-1..1, DY:-1..1.
Neural Convolutional layer

\[ h_{i,j} = \sigma \left( \sum_{m\in[-1,1], n\in[-1,1]} i_{m+i,n+j} \cdot w_{m,n} \right) \]

\[ \sigma(X) := \frac{1}{1 + \exp(-X)}. \]

= \sigma(activation(I, J))

activation(I, J) += input(I + M, J + N) * weight(M, N).

weight(DX, DY) := random(*, -1, 1) for DX:-1..1, DY:-1..1.

Summation became an aggregator
Neural Convolutional layer

\[ h_i, j := \sigma(\sigma(\text{activation}(I, J)) + \text{input}(I + M, J + N) \ast \text{weight}(M, N)) \]

\[ \sigma(\sigma(X)) := \frac{1}{1 + \exp(-X)} \]

\[ = \sigma(\text{activation}(I, J)) \]

\[ \text{activation}(I, J) += \text{input}(I + M, J + N) \ast \text{weight}(M, N) \]

\[ \text{weight}(DX, DY) := \text{random}(*,-1,1) \text{ for } DX:-1..1, DY:-1..1. \]
Neural Convolutional layer

\[ h_{i,j} = \sigma \left( \sum_{m \in [-1,1], n \in [-1,1]} i_{m+i,n+j} \ast w_{m,n} \right) \]

\[ \sigma\sigma(X) := \frac{1}{1 + \exp(-X)}. \]
\[ = \sigma(\text{activation}(I, J)). \]
\[ \text{activation}(I, J) += \text{input}(I + M, J + N) \ast \text{weight}(M, N). \]
\[ \text{weight}(DX, DY) := \text{random}(*,-1,1) \text{ for } DX:-1..1, DY:-1..1. \]

Our ranges over \( m \) and \( n \) are reflected in the shape of weight.
Neural Convolutional layer

\[ h_{i,j} = \sigma \left( \sum_{m \in [-1,1], n \in [-1,1]} i_{m+i, n+j} \ast w_{m,n} \right) \]

\[ \sigma(\Sigma) := 1 / (1 + \exp(-\Sigma)) \]

= \sigma(activation(I, J)).

activation(I, J) = input(I + M, J + N) \ast weight(M, N).

weight(DX,DY) := random(*,-1,1) for DX:-1..1, DY:-1..1.

Here keys are integers but we can also support more complicated structures.
More Neural

\[ \sigma(X) = \frac{1}{1 + \exp(-X)}. \]
\[ \text{out}(J) = \sigma(\text{activation}(J)). \]
\[ \text{activation}(J) += \text{out}(I) * \text{edge}(I,J). \]
More Neural

\[ \sigma(X) = \frac{1}{1 + \exp(-X)}. \]
\[ \text{out}(J) = \sigma(\text{activation}(J)). \]
\[ \text{activation}(J) += \text{out}(I) \ast \text{edge}(I,J). \]

All weights have been rolled into the edges connecting neurons.
More Neural

\( \sigma(X) = \frac{1}{1 + \exp(-X)}. \)

\( \text{out}(J) = \sigma(\text{activation}(J)). \)

\( \text{activation}(J) += \text{out}(I) \times \text{edge}(I,J). \)

The output of a neuron is our nonlinearity applied to the sum of its inputs.
More Neural

$$\sigma(X) = \frac{1}{1 + \exp(-X)}.$$  
$$\text{out}(J) = \sigma(\text{activation}(J)).$$  
$$\text{activation}(J) += \text{out}(I) \times \text{edge}(I,J).$$

Note: nowhere in this program do we specify the form of our variables $I$, $J$. 
More Neural

\[ \sigma(X) = \frac{1}{1 + \exp(-X)}. \]
out(J) = \sigma(activation(J)).
activation(J) += out(I) * edge(I,J).

\[
\text{edge(input}(X,Y),\text{hidden}(X+DX,Y+DY)) = \text{weight}(DX,DY).
\text{weight}(DX,DY) := \text{random}(*,-1,1) \text{ for } DX:-1..1,DY:-1..1.
\]

Instead we can specify the structure of keys inside the definition of edge.
More Neural

\[ \sigma(X) = \frac{1}{1 + \exp(-X)}. \]
\[ \text{out}(J) = \sigma(\text{activation}(J)). \]
\[ \text{activation}(J) \leftarrow \text{out}(I) \times \text{edge}(I,J). \]

\[ \text{edge}(\text{input}(X,Y),\text{hidden}(X+DX,Y+DY)) = \text{weight}(DX,DY). \]
\[ \text{weight}(DX,DY) := \text{random}(*,-1,1) \text{ for } DX:-1..1, DY:-1..1. \]

The weight do not depend on the absolute location of the input, so this is a convolution again.
Dyna Makes Algorithms Easy
Dyna Makes Algorithms Easy

• Gibbs, MCMC – flip variable, compute likelihood ratio, accept or reject
Dyna Makes Algorithms Easy

• Gibbs, MCMC – flip variable, compute likelihood ratio, accept or reject
• Iterative algorithms – loopy belief propagation, numerical optimization
Dyna Makes Algorithms Easy

• Gibbs, MCMC – flip variable, compute likelihood ratio, accept or reject
• Iterative algorithms – loopy belief propagation, numerical optimization
• Neural networks – computation graphs passing dense matrices
Dyna Makes Algorithms Easy

• Gibbs, MCMC – flip variable, compute likelihood ratio, accept or reject
• Iterative algorithms – loopy belief propagation, numerical optimization
• Neural networks – computation graphs passing dense matrices
• Branch-and-bound, Davis–Putnam–Logemann–Loveland (DPLL)
Dyna Makes Algorithms Easy

- Gibbs, MCMC – flip variable, compute likelihood ratio, accept or reject
- Iterative algorithms – loopy belief propagation, numerical optimization
- Neural networks – computation graphs passing dense matrices
- Branch-and-bound, Davis–Putnam–Logemann–Loveman (DPLL)
Dyna Makes Algorithms Easy

• Gibbs, MCMC – flip variable, compute likelihood ratio, accept or reject
• Iterative algorithms – loopy belief propagation, numerical optimization
• Neural networks – computation graphs passing dense matrices
• Branch-and-bound, Davis–Putnam–Logemann–Loveland (DPLL)

• Implementations and more in:
  • Dyna: Extending Datalog for modern AI. (Eisner & Filardo 2011)
  • Dyna: A non-probabilistic language for probabilistic AI. (Eisner 2009)
How much can a declarative language save us?
Implementing shortest path
Implementing shortest path

Dyna (Declarative)

distance(X) min= edge(X, Y)
    + distance(Y).
distance(start) min= 0.
path_length = distance(end).
Implementing shortest path

Dyna (Declarative)

\[ \text{distance}(X) \min = \text{edge}(X, Y) + \text{distance}(Y). \]
\[ \text{distance}(\text{start}) \min = 0. \]
\[ \text{path\_length} = \text{distance}(\text{end}). \]

Java (Procedural)

```java
queue = new FifoQueue<Pair<String, Float>>();
distances = new HashMap<String, Float>();
edges = new HashMap<Pair<String, String>, Float>();
// load edges
queue.push("start");
while(!queue.empty()) {
    d = queue.pop();
    for (e : edge) {
        if (e.first().second().equals(d.first())) {
            if (distance.get(e.first()) <
                d.second() + e.second()) {
                distance.put(e.first(),
                    d.second() + e.second());
                queue.push(e.first());
            }
        }
    }
}
path_length = distances.get("end");
```
Implementing shortest path

Dyna (Declarative)

distance(X) min= edge(X, Y) + distance(Y).
distance(start) min= 0.
path_length = distance(end).

Java (Procedural)

```java
queue = new PriorityQueue<String, Float>();
distances = new HashMap<String, Float>();
edges = new HashMap<Pair<String, String>, Float>();
// load edges
queue.push("start", 0);
while(!queue.empty()) {
    d = queue.pop();
    for(e : edge) {
        if(e.first().second().equals(d.first())) {
            n = d.second() + e.second();
            if(distance.get(e.first()) < n) {
                distance.put(e.first(), n);
                queue.push(e.first(), n);
            }
        }
    }
}
path_length = distances.get("end");
```
Implementing shortest path

Dyna (Declarative)

\[
\text{distance}(X) \text{ min} = \text{edge}(X, Y) + \text{distance}(Y).
\]
\[
\text{distance}(\text{start}) \text{ min} = 0.
\]
\[
\text{path\_length} = \text{distance}(\text{end}).
\]

Java (Procedural)

```java
queue = new PriorityQueue<String, Float>();
distances = new HashMap<String, Float>();
edges = new HashMap<String, Map<String, Float>>();
// load edges
queue.push("start", 0);
while(!queue.empty()) {
    d = queue.pop();
    for(e : edge.get(d.first())) {
        n = d.second() + e.second();
        if(distance.get(e.first()) < n) {
            distance.put(e.first(), n);
            queue.push(e.first(), n);
        }
    }
}
path_length = distances.get("end");
```
Implementing shortest path

Dyna (Declarative)

distance(X) min= edge(X, Y) + distance(Y).
distance(start) min= 0.
path_length = distance(end).

Java (Procedural)

```java
placeIndex = new HashMap<String, Integer>();
queue = new PriorityQueue<Integer, Float>();
distances = new float[num_places];
edges = new float[num_places][num_places];
// load edges
queue.push(placesIndex.get("start"), 0);
while(!queue.empty()) {
    d = queue.pop();
    l = edges[d.first()];
    for(j = 0; j < l.length; j++) {
        n = d.second() + l[j];
        if(distances[j] < n) {
            distances[j] = n;
            queue.push(j, n);
        }
    }
}
path_length = distances[placeIndex.get("end")];
```
Implementing shortest path

Dyna (Declarative)

distance(X) min= edge(X, Y) + distance(Y).
distance(start) min= 0.
path_length = distance(end).

Java (Procedural)

placeIndex = new HashMap<String, Integer>();
edges = new float[num_places][num_places];
// load edges
float distance(from) {
    if(from == placeIndex.get("start")) {
        return 0;
    }
    l = edges[from];
    r = infinity;
    for(j = 0; j < l.length; j++) {
        n = distance(j) + l[j];
        if(n < r)
            r = n;
    }
    return r;
}
path_length = distance(placeIndex.get("end"));
Implementing shortest path

**Dyna (Declarative)**

\[
\text{distance}(X) \ \text{min} = \ \text{edge}(X, Y) + \ \text{distance}(Y).
\]
\[
\text{distance}(\text{start}) \ \text{min} = 0.
\]

**Java (Procedural)**

```java
placeIndex = new HashMap<String, Integer>();
edges = new float[num_places][num_places];
// load edges
float distance(from) {
    if (from == placeIndex.get("start")) {
        return 0;
    }
    l = edges[from];
    r = infinity;
    for (j = 0; j < l.length; j++) {
        n = distance(j) + l[j];
        if (n < r)
            r = n;
    }
    return r;
}

path_length = distance(placeIndex.get("end"));
```

A single Dyna program can represent hundreds of possible implementations.

Other implementations (not shown here) include A* and bidirectional search, and choice of data structures to support dynamic graphs.
Given all of these implementations,
Given all of these implementations, the problem is choice.

If you are Neo, you have two choices
If you are Neo, you have two choices

• Take the Architect’s deal
  • Restart the Matrix
  • Let all the humans in Zion die
    • But restart Zion with 16 females and 7 males (fight another day)
If you are Neo, you have two choices

• Take the Architect’s deal
  • Restart the Matrix
  • Let all the humans in Zion die
    • But restart Zion with 16 females and 7 males (fight another day)
  • Already tried this 5 times
If you are Neo, you have two choices

• Take the Architect’s deal
  • Restart the Matrix
  • Let all the humans in Zion die
    • But restart Zion with 16 females and 7 males (fight another day)
• Already tried this 5 times
• Current argmax
If you are Neo, you have two choices

• Take the Architect’s deal
  • Restart the Matrix
  • Let all the humans in Zion die
    • But restart Zion with 16 females and 7 males (fight another day)
  • Already tried this **5 times**
  • Current argmax

• Follow “an emotion specifically designed to overwhelm logic & reason”
  • Save Trinity
If you are Neo, you have two choices

• Take the Architect’s deal
  • Restart the Matrix
  • Let all the humans in Zion die
    • But restart Zion with 16 females and 7 males (fight another day)
  • Already tried this 5 times
  • Current argmax

• Follow “an emotion specifically designed to overwhelm logic & reason”
  • Save Trinity
  • YOLO, figure this out as we go (unknown reward)
This raises the next question ...
This raises the next question ... can machines love
This raises the next question ... can machines love or at least make irrational choices
This raises the next question ... can machines love or at least make irrational choices

\[ \text{action} = \text{argmax} \pi(\cdot | \ldots) \]

The “Rational” Choice
This raises the next question …

can machines love

or at least make irrational choices

\[
\text{action} = \text{argmax } \pi(\cdot|\ldots)
\]

The “Rational” Choice

\[
\text{action} \sim \pi(\cdot|\ldots)
\]

The “Irrational” Choice

(Randomly sample)
This raises the next question ... can machines love or at least make irrational choices

\[ action = \text{argmax} \, \pi(\cdot | \ldots) \]

The “Rational” Choice

\[ action \sim \pi(\cdot | \ldots) \]

The “Irrational” Choice (Randomly sample)

Exploitation

Exploration
Outline

• Why Declarative Programming?
• Quick introduction to the Dyna language
• Automatic optimization of Dyna programs
Dyna source code (micro-example)

```plaintext
output max= input(I).
input("a") := 1.
input("b") := 2.
```
Dyna

Dynamic data structure

output = 2
output = 1

input("b") := 0
output

query
update
query

response

compile

Dyna source code
(micro-example)

output max= input(I).
input("a") := 1.
input("b") := 2.
**Dyna**

Dynamic data structure

- **input("b") := 0**
- **output**
- **query**
- **update**
- **query**
- **response**
- **output = 2**
- **response**
- **output = 1**

**Dyna source code**

```
output max= input(I).
input("a") := 1.
input("b") := 2.
```
**Dyna**

Dynamic data structure

Dyna source code (micro-example)

```
output max= input(I).
input("a") := 1.
input("b") := 2.
```
Dyna

Dynamic data structure

compile

Dyna source code (micro-example)

output \text{max} = \text{input(I)}.
input("a") := 1.
input("b") := 2.
Dyna

Dynamic data structure

input("b") := 0
output

query

update

query

response

output = 2

response

output = 1

Dyna source code (micro-example)

output max= input(I).
input("a") := 1.
input("b") := 2.
Dyna

output = 2
output = 1

Dynamic data structure

input("b") := 0
output

query
update
query
response
response

compile

Dyna source code (micro-example)

output max= input(I).
input("a") := 1.
input("b") := 2.
Dyna

Dynamic data structure

output = 2

input("b") := 0
output

query
update
query
response
output = 1
response
output = 1

Dyna source code (micro-example)

```plaintext
output max= input(I).
input("a") := 1.
input("b") := 2.
```
Tuning Dyna

![Diagram showing the interaction between Workload and Dynamic data structure]

- **Workload**
- **Dynamic data structure**
  - **Update**
  - **Query**
  - **Response (via callback)**
Tuning Dyna

Workload → Dynamic data structure

- Update
- Query

Response (via callback)

Latency
Tuning Dyna

Solver has “knobs” to tune

Workload

Dynamic data structure

query

update

response (via callback)

latency
Tuning Dyna

Solver has “knobs” to tune
Tuning Dyna

Solver has “knobs” to tune

Example knob: eager or lazy updates?
e.g., dynamic max data structure

```matlab
% Dyna:
output max = input(I).
```

Max-heap:
- $O(\log n)$ per update
- $O(n)$ per batch update (“heapify”)
Tuning Dyna

Example knob: eager or lazy updates? e.g., dynamic max data structure

% Dyna:
output max = input(I).

Max-heap:
- $O(\log n)$ per update
- $O(n)$ per batch update (“heapify”)
Tuning Dyna

Example knob: eager or lazy updates? e.g., *dynamic max data structure*

\[
\rho(\pi) = \mathbb{E} \left[ \sum_{i=1}^{\infty} \gamma^i \lambda_i \text{latency}(i) \right]
\]

Max-heap:
- \(O(\log n)\) per update
- \(O(n)\) per batch update ("heapify")
Tuning Dyna

**Dynamic data structure**

Example knob: eager or lazy updates?

- e.g., *dynamic max data structure*

\[ \rho(\pi) = \mathbb{E} \left[ \sum_{i=1}^{\infty} \gamma^i \lambda_i \text{latency}(i) \right] \]

Max-heap:
- \(O(\log n)\) per update
- \(O(n)\) per batch update (“heapify”)

Solver has “knobs” to tune

- Total cost knob setting
- Average latency on workload

Knob settings
Tuning Dyna

Solver has “knobs” to tune

Example knob: eager or lazy updates?
e.g., dynamic max data structure

\% Dyna:
output \text{max} = \text{input}(I).

Max-heap:
- $O(\log n)$ per update
- $O(n)$ per batch update (“heapify”)
Tuning Dyna

Workload \rightarrow \text{Dynamic data structure} \rightarrow \text{response (via callback)}

- **Example knob:** eager or lazy updates?
- **Dynamic max data structure**

\[ \rho(\pi) = \mathbb{E}\left[ \sum_{i=1}^{\infty} \gamma^i \lambda_i \text{latency}(i) \right] \]

- **Total cost knob setting**
- **Encourage earlier jobs to finish first**

Max-heap:
- \( O(\log n) \) per update
- \( O(n) \) per batch update ("heapify")

Solver has “knobs” to tune

- **input:** \( \text{max} = \text{input}(I) \)

**Optimizer**

Example knob: eager or lazy updates? e.g., *dynamic max data structure*
Off-line training

Fiddle with knobs
Off-line training

Fiddle with knobs

Run entire workload
Off-line training

Fiddle with knobs

Run entire workload

Feedback $\rho(\pi)$
Off-line training

- Reasonable way to tune knobs off-line (used by PhiPac, ATLAS, SATZilla)
Off-line training

- Reasonable way to tune knobs off-line (used by PhiPac, ATLAS, SATZilla)
Off-line training

- Reasonable way to tune knobs off-line (used by PhiPac, ATLAS, SATZilla)
- The inefficiency: this loop *explores* one policy per run.
Off-line training

- Reasonable way to tune knobs off-line (used by PhiPac, ATLAS, SATZilla)
- The inefficiency: this loop *explores* one policy per run.
- How do we tighten the loop to get feedback more often?
Off-line training

- Fiddle with knobs
- Run entire workload
- Feedback $\rho(\pi)$

- Reasonable way to tune knobs off-line (used by PhiPac, ATLAS, SATZilla)
- The inefficiency: this loop explores one policy per run.
- How do we tighten the loop to get feedback more often?
Inside the solver

queries & updates

response
Inside the solver

queries & updates → agenda → response
Inside the solver

queries & updates → agenda → thread → response
Inside the solver

queries & updates
agenda
pop
response

thread
Inside the solver

% matrix multiplication
\[ c(I,K) += a(I,J) \times b(J,K). \]

Task:
Compute \( c(I,4) \) for all \( I \)

queries & updates

agenda

pop

response

task

thread
% matrix multiplication
\[ c(I,K) += a(I,J) \times b(J,K). \]

Task:
Compute \( c(I,4) \) for all \( I \)

Inside the solver

queries & updates → agenda → pop → task → dispatch → strategy

thread

response
Inside the solver

% matrix multiplication
\[
c(I,K) \ += \ a(I,J) \ * \ b(J,K).
\]

Task:
Compute \( c(I,4) \) for all \( I \)

Strategy:
for \( J \) in \( b(:,4) \):
    for \( I \) in \( a(:,J) \):
        \( c(I,4) \ += \ a(I,J) \ * \ b(:,4) \)

queries & updates

agenda

pop

task

dispatch

strategy

thread

response
Inside the solver

% matrix multiplication
\[ c(I, K) += a(I, J) \times b(J, K). \]

Task:
Compute \( c(I, 4) \) for all \( I \)

Strategy:
for \( J \) in \( b(:, 4) \):
    for \( I \) in \( a(:, J) \):
        \[ c(I, 4) += a(I, J) \times b(:, 4) \]

queries & updates

agenda

pop

task

strategy

new tasks

computed values

run

thread

response

"New tasks" are computed values

"Tasks" are response to queries & updates
Inside the solver

Task: Compute $c(I, 4)$ for all $I$

Strategy:
for J in $b(:, 4)$:
for I in $a(:, J)$:
$c(I, 4) += a(I, J) \times b(:, 4)$

% matrix multiplication
$c(I, K) += a(I, J) \times b(J, K)$.

queries & updates

agenda

pop

new tasks

computed values

run

strategy

thread

task

response
Inside the solver

\[
\begin{align*}
\text{Task: Compute } c(I, 4) \text{ for all } I \\
\text{Strategy:} \\
&\text{for } J \text{ in } b(:, 4): \\
&\text{for } I \text{ in } a(:, J): \\
&c(I, 4) += a(I, J) \times b(:, 4)
\end{align*}
\]

% matrix multiplication
\[
c(I, K) += a(I, J) \times b(J, K).
\]
Inside the solver

% matrix multiplication
\[ c(I,K) += a(I,J) \times b(J,K). \]

Task:
Compute \( c(I,4) \) for all \( I \)

Strategy:
for \( J \) in \( b(:,4) \):
   for \( I \) in \( a(:,J) \):
      \[ c(I,4) += a(I,J) \times b(:,4) \]

queries & updates
agenda
push
push
new tasks
run
strategy
dispatch
pop

task

computed values
memoize
lookup

thread

response

cache
Inside the solver

Tasks on agenda can be executed in any order! Interpolates between eager and lazy strategies.

% matrix multiplication
\[ c(I,K) += a(I,J) \times b(J,K). \]

Task:
Compute \( c(I,4) \) for all \( I \)

Strategy:
\[
\text{for } J \text{ in } b(:,4):
    \text{for } I \text{ in } a(:,J):
        c(I,4) += a(I,J) \times b(:,4)
\]

Tasks on agenda can be executed in any order! Interpolates between eager and lazy strategies.

queries & updates

agenda

pop

task

dispatch

push

new tasks

run

strategy

memoize

computed values

lookup

cache

thread

response

Task:
Compute \( c(I,4) \) for all \( I \)

Strategy:
\[
\text{for } J \text{ in } b(:,4):
    \text{for } I \text{ in } a(:,J):
        c(I,4) += a(I,J) \times b(:,4)
\]
Inside the solver

% matrix multiplication
\[ c(I, K) += a(I, J) \times b(J, K). \]

Task:
Compute \( c(I, 4) \) for all \( I \)

Strategy:
for \( J \) in \( b(:, 4) \):
  for \( I \) in \( a(:, J) \):
    \( c(I, 4) += a(I, J) \times b(:, 4) \)

Tasks on agenda can be executed in any order!
Interpolates between eager and lazy strategies

queries & updates

agenda

choose!
pop

new
tasks

run

strategy

dispatch

memoize

computed
values

push

response

cache

lookup

thread

Tasks on agenda can be executed in any order!
Interpolates between eager and lazy strategies
Inside the solver

% matrix multiplication
\[ c(I, K) \ += \ a(I, J) \ * \ b(J, K). \]

Strategy:

\[
\text{for } J \text{ in } b(:,4):
\text{for } I \text{ in } a(:,J):
\quad c(I,4) \ += \ a(I,J) \ * \ b(:,4)
\]

Tons of admissible strategies.
Each attempts to make progress toward an answer to open queries

Tasks on agenda can be executed in any order!
Interpolates between eager and lazy strategies

Queries & updates
Inside the solver

Tasks on agenda can be executed in any order! Interpolates between eager and lazy strategies.

Queries & updates

Agenda

Task

Strategy:

for J in b(:,4):
  for I in a(:,J):
    c(I,4) += a(I,J) * b(:,4)

Tons of admissible strategies. Each attempts to make progress toward an answer to open queries.

New tasks

Computed values

Memoize

Lookup

Thread

Response

Matrix multiplication:

\[ c(I,K) \ += \ a(I,J) \ * \ b(J,K). \]
% matrix multiplication
\[ c(I, K) += a(I, J) \times b(J, K). \]

Tasks on agenda can be executed in any order!
Interpolates between eager and lazy strategies

Strategy:
for J in b(:, 4):
    for I in a(:, J):
        c(I, 4) += a(I, J) \times b(:, 4)

Tons of admissible strategies.
Each attempts to make progress toward an answer to open queries

Tasks on agenda can be executed in any order!
Interpolates between eager and lazy strategies

Memos are optional.
Solver can create or flush memos anytime.
(memos save recomputation, but require maintenance)

Memos are optional.
Solver can create or flush memos anytime.
(memos save recomputation, but require maintenance)
Tasks on agenda can be executed in any order! Interpolates between eager and lazy strategies.

Memos are optional. Solver can create or flush memos anytime. (memos save recomputation, but require maintenance)

strategy

Strategy:
for J in b(:,4):
    for I in a(:,J):
        c(I,4) += a(I,J) * b(:,4)

Tasks on agenda can be executed in any order! Interpolates between eager and lazy strategies.

% matrix multiplication
c(I,K) += a(I,J) * b(J,K).

Tons of admissible strategies. Each attempts to make progress toward an answer to open queries.

response

Task:
Compute c(I,4) for all I

Memos are optional. Solver can create or flush memos anytime. (memos save recomputation, but require maintenance)
Strategy options
Strategy options

Solver should systematize all the reasonable implementation tricks that programmers might use and make them work together correctly.
Strategy options

Solver should systematize all the reasonable implementation tricks that programmers might use and make them work together correctly.

- Parallelizing independent computations
Strategy options

Solver should systematize all the reasonable implementation tricks that programmers might use and make them work together correctly.

- Parallelizing independent computations
- Ordering dependent computations
Strategy options

Solver should systematize all the reasonable implementation tricks that programmers might use and make them work together correctly.

- **Parallelizing independent computations**
- **Ordering dependent computations**
  - Join strategies
Strategy options

Solver should systematize all the reasonable implementation tricks that programmers might use and make them work together correctly.

- **Parallelizing independent computations**
- **Ordering dependent computations**
  - Join strategies
  - Forward vs. backward chaining (update-driven vs. query-driven)
Strategy options

Solver should systematize all the reasonable implementation tricks that programmers might use and make them work together correctly.

- **Parallelizing independent computations**
- **Ordering dependent computations**
  - Join strategies
  - Forward vs. backward chaining (update-driven vs. query-driven)
  - Dynamically identify unnecessary computation
Strategy options

Solver should systematize all the reasonable implementation tricks that programmers might use and make them work together correctly.

- **Parallelizing independent computations**
- **Ordering dependent computations**
  - Join strategies
  - Forward vs. backward chaining (update-driven vs. query-driven)
  - Dynamically identify unnecessary computation
    - Short-circuiting, branch-and-bound/A*, watched variables
Strategy options

Solver should systematize all the reasonable implementation tricks that programmers might use and make them work together correctly.

- **Parallelizing independent computations**
- **Ordering dependent computations**
  - Join strategies
  - Forward vs. backward chaining (update-driven vs. query-driven)
  - Dynamically identify unnecessary computation
    - Short-circuiting, branch-and-bound/A*, watched variables
- **Consolidating related work**
Strategy options

Solver should systematize all the reasonable implementation tricks that programmers might use and make them work together correctly.

- **Parallelizing independent computations**
- **Ordering dependent computations**
  - Join strategies
  - Forward vs. backward chaining (update-driven vs. query-driven)
  - Dynamically identify unnecessary computation
    - Short-circuiting, branch-and-bound/A*, watched variables
- **Consolidating related work**
  - Static or dynamic batching (consolidating similar tasks, including GPU)
Strategy options

Solver should systematize all the reasonable implementation tricks that programmers might use and make them work together correctly.

- **Parallelizing independent computations**
- **Ordering dependent computations**
  - Join strategies
  - Forward vs. backward chaining (update-driven vs. query-driven)
  - Dynamically identify unnecessary computation
    - Short-circuiting, branch-and-bound/A*, watched variables
- **Consolidating related work**
  - Static or dynamic batching (consolidating similar tasks, including GPU)
  - Inlining depth (consolidating caller-callee)
Strategy options

Solver should systematize all the reasonable implementation tricks that programmers might use and make them work together correctly.

- **Parallelizing independent computations**
- **Ordering dependent computations**
  - Join strategies
  - Forward vs. backward chaining (update-driven vs. query-driven)
  - Dynamically identify unnecessary computation
    - Short-circuiting, branch-and-bound/A*, watched variables
- **Consolidating related work**
  - Static or dynamic batching (consolidating similar tasks, including GPU)
  - Inlining depth (consolidating caller-callee)
- **Storage**
Strategy options

Solver should systematize all the reasonable implementation tricks that programmers might use and make them work together correctly.

- **Parallelizing independent computations**
- **Ordering dependent computations**
  - Join strategies
  - Forward vs. backward chaining (update-driven vs. query-driven)
  - Dynamically identify unnecessary computation
    - Short-circuiting, branch-and-bound/A*, watched variables
- **Consolidating related work**
  - Static or dynamic batching (consolidating similar tasks, including GPU)
  - Inlining depth (consolidating caller-callee)
- **Storage**
  - Memoization policy; choose low-level data structures
The Dyna solver
The Dyna solver

• Lots of challenges in defining this giant space while preserving correctness
The Dyna solver

- Lots of challenges in defining this giant space while preserving correctness
- Most systems avoid choices. We embrace choice because we have machine learning to choose intelligently.
The Dyna solver

• Lots of challenges in defining this giant space while preserving correctness
• Most systems avoid choices. We embrace choice because we have machine learning to choose intelligently.
• Further reading
  • Mixed-chaining / arbitrary memoization (Filardo & Eisner, 2012)
  • Set-at-a-time inference (Filardo & Eisner, 2017, in preparation)
The Dyna solver

• Lots of challenges in defining this giant space while preserving correctness
• Most systems avoid choices. We embrace choice because we have machine learning to choose intelligently.
• Further reading
  • Mixed-chaining / arbitrary memoization (Filardo & Eisner, 2012)
  • Set-at-a-time inference (Filardo & Eisner, 2017, in preparation)
• Lots of progress to come!
The Dyna solver

• Lots of challenges in defining this giant space while preserving correctness

• Most systems avoid choices. We embrace choice because we have machine learning to choose intelligently.

• Further reading
  • Mixed-chaining / arbitrary memoization (Filardo & Eisner, 2012)
  • Set-at-a-time inference (Filardo & Eisner, 2017, in preparation)

• Lots of progress to come!
  • Nathaniel Wesley Filardo is tying up many loose ends in his thesis (September 2017)
The Dyna solver

• Lots of challenges in defining this giant space while preserving correctness
• Most systems avoid choices. We embrace choice because we have machine learning to choose intelligently.

• Further reading
  • Mixed-chaining / arbitrary memoization (Filardo & Eisner, 2012)
  • Set-at-a-time inference (Filardo & Eisner, 2017, in preparation)

• Lots of progress to come!
  • Nathaniel Wesley Filardo is tying up many loose ends in his thesis (September 2017)
  • He’s on the job market!
Sequential choices at runtime (some stochastic)
Sequential choices at runtime (some stochastic)

edge (...) query
Sequential choices at runtime (some stochastic)
Sequential choices at runtime (some stochastic)

edge(...)
query

edge(x, x)

...
Sequential choices at runtime (some stochastic)
Sequential choices at runtime (some stochastic)

edge(...) query

edge(x, Y) for some x

answer from edge rules using join strategy 1

join strategy 2
Sequential choices at runtime (some stochastic)

**Query**: edge($\ldots$)

for some $x$

query edge($x$, $Y$)

answer will be an iterator over outgoing edges (adjacency list)

answer from edge rules using join strategy 1

join strategy 2

**Diagram**:
- Edge rules for stochastic behavior
  - rand() ≤ 0.3
  - otherwise
Sequential choices at runtime (some stochastic)

query edge \((x, Y)\) for some \(x\)

answer will be an iterator over outgoing edges (adjacency list)

hash \((x) / H\) \leq 0.2

look up answer in dense array \(A4\) indexed by \(x\) (where \(x\) is an integer or is represented as one)

answer from edge rules using join strategy 1

join strategy 2
Sequential choices at runtime (some stochastic)

- `edge(...) query` for some `x`
- `edge(x, y)`
- `hash(x) / H ≤ 0.2`
- Look up answer in dense array A4 indexed by `x` (where `x` is an integer or is represented as one)
- `look for answer in sparse hash table H5`
- `answer from edge rules using join strategy 1`
- `join strategy 2`
- `answer will be an iterator over outgoing edges (adjacency list)`
- `rand() ≤ 0.3`
Sequential choices at runtime (some stochastic)

edge(...) query

edge(x, y) for some x

hash(x) / H ≤ 0.2

look up answer in dense array A4 indexed by x (where x is an integer or is represented as one)

look for answer in sparse hash table H5

answer from edge rules using join strategy 1

join strategy 2

answer will be an iterator over outgoing edges (adjacency list)

cached

return answer
Sequential choices at runtime (some stochastic)

edge(\ldots)

query

for some \( x \)

edge(\( x, Y \))

answer will be an iterator over outgoing edges (adjacency list)

\text{hash}(x) / H \leq 0.2

look up answer in dense array A4 indexed by \( x \) (where \( x \) is an integer or is represented as one)

\text{hash}(x)/H

otherwise

look for answer in sparse hash table H5

\text{join strategy 1}

\text{join strategy 2}

\text{cached}

otherwise

return answer

query edge(\( X, Y \)) and filter to \( X = x \)
Sequential choices at runtime (some stochastic)

- For some \( x \) and \( Y \), the edge query \( \text{edge}(x, Y) \) will be an iterator over outgoing edges (adjacency list) if \( \text{hash}(x)/H \leq 0.2 \).
  - Look up answer in dense array \( A_4 \) indexed by \( x \) (where \( x \) is an integer or is represented as one).
  - Look for answer in sparse hash table \( H_5 \) using join strategy 1 or 2.
- Answer from edge rules.
- Using join strategy 1.
- Join strategy 2.
- Return answer.
- Memoize & return.
Sequential choices at runtime (some stochastic)

- edge(...) query
- edge(x, y) for some x
- edge(x, x)

answer will be an iterator over outgoing edges (adjacency list)

hash(x) / H \leq 0.2

look up answer in dense array A4 indexed by x (where x is an integer or is represented as one)

- otherwise
- hash(x) / H > 0.2

look for answer in sparse hash table H5

- join strategy 1
- join strategy 2

- cached
- answer from edge rules using join strategy 1
- otherwise
- answer from edge rules using join strategy 2

Query edge(X, Y)

- and filter to X = x

- return answer
- otherwise
- memoize & return
- otherwise
- rand() \leq 0.3

Policy probabilities that are tuned over time (typically approaching 0 or 1)
Policy probabilities can be sensitive to context of task
Policy probabilities can be sensitive to context of task

Stochastically conditioned on the following information (features).
Policy probabilities can be sensitive to context of task

Stochastically conditioned on the following information (features).

• **Task characteristics**
  - What type of task?
  - What are the task parameters?
  - Who requested the task?
Policy probabilities can be sensitive to context of task

Stochastically conditioned on the following information (features).

- **Task characteristics**
  - What type of task?
  - What are the task parameters?
  - Who requested the task?

- **Dataflow**
  - What depends on this task (children)?
  - What does this task depend on (parents)?
Policy probabilities can be sensitive to context of task

Stochastically conditioned on the following information (features).

• **Task characteristics**
  - What type of task?
  - What are the task parameters?
  - Who requested the task?

• **Dataflow**
  - What depends on this task (children)?
  - What does this task depend on (parents)?

• **Agenda characteristics**
  - Are there a lot of open queries?
  - What’s on the agenda? How long has it been there?
Policy probabilities can be sensitive to context of task

Stochastically conditioned on the following information (features).

- **Task characteristics**
  - What type of task?
  - What are the task parameters?
  - Who requested the task?
- **Dataflow**
  - What depends on this task (children)?
  - What does this task depend on (parents)?
- **Agenda characteristics**
  - Are there a lot of open queries?
  - What’s on the agenda? How long has it been there?
- **Cache characteristics**
  - Statistics: number, hit rate, frequency, & recency of memos (broken down by type)
Tuning probabilities
Tuning probabilities

query $\text{edge}(x, y)$
and filter to $x=x$

rand() ≤ 0.1

otherwise

return answer

memoize & return
If we knew the *long-term* cost of each action in context, we could update the policy now!
Tuning probabilities

If we knew the long-term cost of each action in context, we could update the policy now!
Tuning probabilities

query \text{edge}(X, Y)\) and filter to \(X=x\)

If we knew the \textit{long-term} cost of each action in context, we could update the policy now!
Tuning probabilities

query \textit{edge}(X, Y) and filter to \(X=x\)

If we knew the \textit{long-term} cost of each action in context, we could update the policy now!

slow!

hypothetically, fork state and run finish workload
The figure illustrates a probabilistic process with a tree structure. At the root node, there is a decision point with two branches: one labeled with a random variable `rand()` with probability 0.1, and the other with a probability of 0.9 labeled as "otherwise".

- **Branch 1 (rand() ≤ 0.1)**: The process returns the answer.
- **Branch 2 (otherwise)**: The process memoizes and returns.

The figure also includes text that explains the tuning of probabilities. It states:

> If we knew the **long-term** cost of each action in context, we could update the policy now!

This is illustrated by a binary tree with two actions: `a_1` and `a_2`, leading to states `q_1` and `q_2`, respectively.

Additionally, there is a mathematical expression:

\[ \mathbb{E}[q_i] \approx \hat{q}(s, a_i) \]

This suggests using machine learning (ML) to predict future costs.
Use ML to predict future costs!

Generalize from past experience to new situations

$$\mathbb{E} [q_i] \approx \hat{q}(s, a_i)$$

Tuning probabilities

query \textbf{edge}(X, Y)
and filter to $X=x$

If we knew the long-term cost of each action in context, we could update the policy now!

Use ML to predict future costs!
Tuning probabilities

Use ML to predict future costs!

Generalize from past experience to new situations

$\mathbb{E} [q_i] \approx \hat{q}(s, a_i) = w^\top \Phi(s, a_i)$

If we knew the long-term cost of each action in context, we could update the policy now!

Use ML to predict future costs!

Generalize from past experience to new situations

$\mathbb{E} [q_i] \approx \hat{q}(s, a_i) = w^\top \Phi(s, a_i)$

hypothetically, fork state and run finish workload

Slow!
Temporal difference learning
Temporal difference learning

\[ \hat{q}(s, a) \approx \mathbb{E} \left[ r + \hat{q}(s', a') \right] \]
Temporal difference learning

\[ \hat{q}(s, a) \approx \mathbb{E} [r + \hat{q}(s', a')] \]
Temporal difference learning

\[ \hat{q}(s, a) \approx \mathbb{E} [r + \hat{q}(s', a')] \]

Make estimator agree with itself

Approximate dynamic programming
Actor-critic policy gradient

\[ S \]

\[ \hat{q}(s, a_1) \]
\[ \hat{q}(s, a_2) \]
\[ \hat{q}(s, a_3) \]
\[ \hat{q}(s, a_4) \]
Actor-critic policy gradient

\[ \pi(a|s) \propto \exp (\theta^T f(s, a)) \]
Actor-critic policy gradient

\[ \pi(a|s) \propto \exp(\theta^T f(s, a)) \]

\[ \theta \leftarrow \theta - \alpha_t \cdot \hat{q}(s, a) \nabla \log \pi(a|s) \]

Says increase the probability of lower cost actions.
Actor-critic policy gradient

\[ \pi(a|s) \propto \exp (\theta^T f(s, a)) \]

\[ \theta \leftarrow \theta - \alpha_t \cdot \hat{q}(s, a) \nabla \log \pi(a|s) \]

Says increase the probability of lower cost actions.
Actor-critic policy gradient

\[ \pi(a|s) \propto \exp(\theta^T f(s, a)) \]

\[ \theta \leftarrow \theta - \alpha_t \cdot \hat{q}(s, a) \nabla \log \pi(a|s) \]

\[ w = w - \beta_t \cdot (\hat{q}(s, a) - (r + \hat{q}(s', a')) \cdot \Phi(s, a) \]

Says increase the probability of lower cost actions.
Summary
Summary

• Declarative languages lend themselves to automated optimization because their solvers have a lot of freedom.
Summary

• Declarative languages lend themselves to automated optimization because their solvers have a lot of freedom.
• Tuning such as solver is a sequential decision making problem that can be tuned with online reinforcement learning techniques.
Summary

• Declarative languages lend themselves to automated optimization because their solvers have a lot of freedom.

• Tuning such as solver is a sequential decision making problem that can be tuned with online reinforcement learning techniques.

• Dyna
  • Has been designed from the ground up be as flexible as possible.
  • A powerful language for specifying machine learning applications.
Summary

• Declarative languages lend themselves to automated optimization because their solvers have a lot of freedom.

• Tuning such as solver is a sequential decision making problem that can be tuned with online reinforcement learning techniques.

• Dyna
  • Has been designed from the ground up be as flexible as possible.
  • A powerful language for specifying machine learning applications.

Check out the paper! Lots of great technical details in the paper that didn't have time to get into.
State of the language
State of the language

- We have two language prototypes
State of the language

• We have to two language prototypes
  • Dyna 1 prototype (2005) was used in Jason’s lab, fueling a dozen NLP papers!
State of the language

• We have two language prototypes
  • Dyna 1 prototype (2005) was used in Jason’s lab, fueling a dozen NLP papers!
  • Dyna 2 prototype (2013) was used for teaching an NLP course to linguists with no programming background.
State of the language

• We have to two language prototypes
  • Dyna 1 prototype (2005) was used in Jason’s lab, fueling a dozen NLP papers!
  • Dyna 2 prototype (2013) was used for teaching an NLP course to linguists with no programming background.
• Both were inefficient because they used too many one-size-fits-all strategies.
Thanks!

Questions? Comments?

Hire Wes Filardo!
http://cs.jhu.edu/~nwf

http://dyna.org

@xtimv
@matthewfl
Loop order, sparse vector product example

\[
\begin{bmatrix}
1 & \Rightarrow & 2.0 \\
9 & \Rightarrow & 0.1 \\
14 & \Rightarrow & 2.3 \\
83 & \Rightarrow & 8.1
\end{bmatrix}
\rightarrow
\begin{bmatrix}
0 & \Rightarrow & 10.2 \\
1 & \Rightarrow & 4.3 \\
2 & \Rightarrow & 7.2 \\
7 & \Rightarrow & 2.3 \\
14 & \Rightarrow & 9.3 \\
\vdots \\
83 & \Rightarrow & 2.3 \\
107 & \Rightarrow & 99
\end{bmatrix}
\]
Join strategies

- Outer loop on \( b(J,4) \), inner loop \( a(I,J) \)
- Outer loop on \( a(I,J) \), inner loop \( b(J,K) \), filter \( K==4 \)
- Sort \( a \) and \( b \) on \( J \) and use Baeza-Yates intersection
- What if we can’t loop over \( b \)? e.g.,
  \[ b(I,K) = X*Y. \]
- In natural language parsing with the CKY algorithm, an unexpected loop order turned out to be 7-9x faster than the loop order presented in most textbooks because of cache locality (Dunlop et al. 2010).

\[
\% \text{ matrix multiplication} \\
c(I,K) += a(I,J) \times b(J,K).
\]
Memoization and data structures

• Do we store results for future use?
  (tradeoff: memos must be kept up-to-date!)
• What data structure?
Batching, answering bigger queries/updates

• Batch pending \( c \) queries.

• Preemptively compute a broader query, \( c(I, K) \)
  Use clever mat-mul alg. (sparse or dense?)
Inlining
• Inline a and/or b queries.
• Bypass agenda and route directly to consumer, e.g.,
  \[ d(I) \max = c(I,K) . \]
• Reduce overhead of method calls
• Reduce overhead of task indirection through agenda
Mixed policies

• Mixed task strategies selection
  • Policy will encounter similar tasks frequently

• Mixed storage strategies
  • e.g., use a hash table, prefix trie, dense array, ...
  • Problem: random choice of strategy might not be consistent
    • E.g. might write to A and read from B (because of randomness)
Storage

Named maps
key -> value

Named of map

edge("a", "b") = 1.
edge("b", "c") = 2.
edge("c", "d") = 5.
pathCost("a", "d") = 8.
pathCost("a", "c") = 3.

Queries

What’s the weight of edge a->b
Outgoing edges from a
Incoming edges to b
List all edges
List all self loops
Edges with weight >= 10

Implementations?

Hash cons, trie (different orders on keys), dense vs. sparse, sorted (what to sort on)

Efficiency depends on
- Frequency of different queries
- Overhead of read/writes
- Sizes
- Lower-level thresholds
Mixed storage solution

hash(edge("a", "b") / 2^32 ~ Uniform(0,1)

Small, temporary data duplication

Hash table

Trie 1->2

Trie 2->1

edge(:any, :any)

pathCost(:any, :any)

edge(:int, :int)

edge(:str, :str)

edge("a", :str)
Mixed storage solution

\[ \text{hash}(\text{edge}(\text{"a", "b"}))/2^{32} \sim \text{Uniform}(0,1) \]
Mixed storage solution

$\text{hash}(\text{edge}("a", "b")) / 2^{32} \sim \text{Uniform}(0,1)$

- Hash table
- Trie 1->2
- Trie 2->1

Small, temporary data duplication
Learning to **choose** a good strategy
Learning to choose a good strategy
Learning to **choose** a good strategy

1. **Bandit**: Each time we execute a task (e.g., compute $c(I, j)$ for an argument $j$):
1. **Bandit**: Each time we execute a task (e.g., compute $c(I, j)$ for an argument $j$):
   - Randomly try one of the available strategies (explore) according to some probability distribution.
Learning to **choose** a good strategy

1. **Bandit**: Each time we execute a task (e.g., compute $c(I, j)$ for an argument $j$):
   - Randomly try one of the available strategies (explore) according to some probability distribution
   - Bias this distribution in favor of strategies with lower measured cost (exploit).
Learning to **choose** a good strategy

1. **Bandit**: Each time we execute a task (e.g., compute $c(I, j)$ for an argument $j$):
   - Randomly try one of the available strategies (explore) according to some probability distribution
   - Bias this distribution in favor of strategies with lower measured cost (exploit).

2. **Contextual bandit**: Allows distribution to depend on task arguments (e.g., $j$) and solver state.
Learning to **choose** a good strategy

1. **Bandit**: Each time we execute a task (e.g., compute $c(I, j)$ for an argument $j$):
   - Randomly try one of the available strategies (explore) according to some probability distribution
   - Bias this distribution in favor of strategies with lower measured cost (exploit).

2. **Contextual bandit**: Allows distribution to depend on task arguments (e.g., $j$) and solver state.

3. **Reinforcement learning**: Accounts for *delayed* costs of actions.
Back to online training...
Back to online training...

Fiddle with policy → Run entire workload

Feedback $\rho(\pi)$
Solver Actions
Solver Actions

Workload
Solver Actions

Workload

Policy actions
Rewrite the objective function
Rewrite the objective function

Workload $i$

Todos:
- Inline in parentheses
- Use underbrace to give it a name
- Add Jason integral picture?
Rewrite the objective function

\[ \rho(\pi) = \mathbb{E} \left[ \sum_{i=1}^{\infty} \gamma^i \lambda_i \text{latency}(i) \right] \]
Rewrite the objective function

\[ \rho(\pi) = \mathbb{E} \left[ \sum_{i=1}^{\infty} \gamma^i \lambda_i \text{latency}(i) \right] \]

Workload \( i \)

Actions \( t \)

TODO inline in parens
use underbrace to give it a name
Add Jason integral picture?
Rewrite the objective function

\[ \rho(\pi) = \mathbb{E} \left[ \sum_{i=1}^{\infty} \gamma^i \lambda_i \text{latency}(i) \right] \]

\[ = \mathbb{E} \left[ \sum_{t=1}^{\infty} \text{load}(t) \cdot (\text{clock}(t+1) - \text{clock}(t)) \right] \]

Rewrite in terms of the policy’s time scale (used in RL)

TODO inline in parens
use underbrace to give it a name

Add Jason integral picture?
Rewrite the objective function

\[ \rho(\pi) = \mathbb{E} \left[ \sum_{i=1}^{\infty} \gamma^i \lambda_i \text{ latency}(i) \right] \]

\[ = \mathbb{E} \left[ \sum_{t=1}^{\infty} \text{load}(t) \cdot (\text{clock}(t+1) - \text{clock}(t)) \right] \]

Each step tries to decrease the load

\[ \text{load}(t) = \sum_{i \in \mathcal{O}(t)} \gamma^i \lambda_i \]

Rewrite in terms of the policy’s time scale (used in RL)

TODO inline in parens use underbrace to give it a name

Add Jason integral picture?
Life of π
Life of $\pi$
Life of $\pi$
Life of $\pi$
Life of $\pi$
Life of $\pi$

\[ \mathbb{E} [q_i] \approx \hat{q}(s, a_i) \]
Life of $\pi$

Use ML to predict future costs!

Generalize from past experience to new situations

$$\mathbb{E} [q_i] \approx \hat{q}(s, a_i)$$
Life of π

Use ML to predict future costs!

Generalize from past experience to new situations

\[ \mathbb{E} [q_i] \approx \hat{q}(s, a_i) = w^\top \Phi(s, a_i) \]
Life of $\pi$

Use ML to predict future costs!

Generalize from past experience to new situations

$$\mathbb{E} [q_i] \approx \hat{q}(s, a_i) = w^\top \Phi(s, a_i)$$

Features:
- Memos (number, hit rate, frequency, recency)
- Pending tasks (number, age)
- Current load (proximity-to-completion heuristics)
- Dataflow graph
  - Will this be used again?
  - Are other jobs blocking on it?
Back to $\pi$
Back to $\pi$
Back to $\pi$
Back to $\pi$
Back to $\pi$

What should $\pi$ do in this state?
What should $\pi$ do in this state?

$$\pi(s) = \arg\min_a \hat{q}(s, a)$$
Back to $\pi$

Predicting the future requires richer features than simply learning to take good actions.

What should $\pi$ do in this state?

$$\pi(s) = \arg\min_a \hat{q}(s, a)$$
Predicting the future requires richer features than simply learning to take good actions. We need $\pi$ to be really fast to execute!
Fiddle with policy

Run entire workload

Feedback $\rho(\pi)$
Fiddle with policy

Run entire workload

Feedback $\rho(\pi)$
Run entire workload

Feedback $\rho(\pi)$

Fiddle with policy
Run entire workload

Feedback $\rho(\pi)$

Fiddle with policy

Feedback

$\widehat{q}(s, a)$

update $\pi$
Run entire workload

Feedback $\rho(\pi)$

Feedback $\hat{q}(s, a)$

update $\pi$  update $\hat{q}$