AI-Systems

Machine Learning Frameworks

Joseph E. Gonzalez
Co-director of the RISE Lab
jegonzal@cs.berkeley.edu

Class Projects

- Project Signup (QR Code) + link on website
  - Please add your project or join a project by **Wednesday**
- Project Teams (~ 3 people per team)
- One Page Project descriptions are due 9/30 at 11:59
  - Title and team members
  - Project description and what is the key problem being studied
  - Discussion of related work
  - Proposed plans for semester
    - 3 weeks until first presentation (initial results)
    - 8 weeks until end of semester project due
- Google Doc (enable commenting so I can comment on it)
  - Submit your project description to this google form.

Objectives For Today

- Historical Evolution of Machine Learning Frameworks
- Declarative (Lazy) vs Imperative (Eager) DSLs
- Automatic Differentiation
- This weeks reading

Historical Context

Early ML / Stats Languages

- S **Data Programming** Languages
  - Developed in **1976** as **Bell Labs** by John Chambers
  - **Replaced Fortran** by providing higher level APIs, graphics
  - Developed **formula syntax** for describing models
  - Eventually replaced by R …

- R **open-source** implementation of S (S-Plus)
  - Developed in **1990's** at University of Auckland
    - Ross Ihaka, Robert Gentleman
  - Like S/S-Plus → Linear algebra abstractions
  - **Rich set of libraries** for statistical analysis
  - Still widely used

- **Matlab** (Matrix Laboratory) – Numerical Computing Sys.
  - Developed in **1970s** at the University of New Mexico by Cleve Moler
  - Designed to **simplify access** to **LINPACK** and **EISPACK**
  - Reasonable **integration with C/Fortran**
  - Rich **graphical interface** with support for graphical programming
    - Simulink
  - Expensive → Octave **limited** open-source version
  - Popular in applied math, engineering, and controls community
  - **Extremely popular in the machine learning community**
    - We would joke that ML people only knew how to program Matlab
- and then it all changed …

Rise of the **Python** Eco-System

- Development of **%pylab**
  - iPython (2001) + SciPy (2001) + Matplotlib (2003) + NumPy (2006)
  - Functions /APIs were like Matlab so easy to transition
  - **Freeeeeee!**
- **Scikit-learn** – basic ML algorithms and models (2007)
  - Started as Google summer of code project → **developed by INRIA**
  - Wide range of standard machine learning techniques
- ~2012 large fraction of ML community Matlab → Python
  - Why?
- Development remained focused on **algorithms libraries**

Machine Learning Libraries

- **LIBLINEAR/LIBSVM** (2008) – **fast algorithms** for fitting linear models and kernelized SVMs
  - Developed at National Taiwan University for (still used in Sklearn)
- **Vowpal Wabbit** (2010?) – **out-of-core** learning for generalized linear models and others
  - Developed by John Langford while at Yahoo!
  - Popular for high-dimensional features
- **Weka** (Java version 1997) – Collection of ML algorithms for Java
  - Developed at the University of Waikato in New Zealand
  - Provided tools for visualizing and analyzing data
- **Xgboost** (2014) – **distributed** boosted decision trees
  - Developed by Tianqi Chen at University of Washington
- Many more …

Distributed Machine Learning Frameworks

- **Mahout** (2009) – **ML algorithms** on Hadoop
  - Early distributed ML library with **"recommender algorithms"**
  - Unable to leverage memory caching
- **GraphLab** (2010) – Framework for **graph structured algorithms**
  - Contained library of algs. (e.g., Gibbs Sampling, LoopyBP, …)
  - Developed new abstractions for distributed graph algs.
- **Spark mllib / SparkML** (2014) – ML algorithms for Spark
  - Leverages memory caching
  - Benefits from work on GraphLab/Sklearn/SystemML

Languages vs Algorithm Libraries

- **Languages** provided support for mathematical operations
  - User still implemented new models and algorithms using fundamental **linear algebra primitives**
- **Libraries of Algorithms** provided individual learning techniques
  - Often specialized to model/technique (fast and easy-to-use)
- Need something in the middle!

Embedded Domain Specific Languages

- Domain specific languages (DSLs) provide **specialized functionality** for a given task
  - Limited functionality → **simplicity** and **optimization**
  - **Example:** SQL → Specialized for data manipulation
- Embedded DSLs are **libraries** or **language extensions** within a general-purpose language tailored to a specific task
  - Combine benefits of DSL and general languages
  - **Example:** linear algebra libraries
- Embedded DSLs have played a significant role in ML
  - Linear Algebra → Pipelines → Differentiable Programs

Machine Learning Pipelines

- Scikit Learn Pipelines (2011)
  - Describes **composition** of feature transformations and models
  - Enables **end-to-end training** and **standardized** prediction

```
steps = [('scaler', StandardScaler()), ('SVM', SVC())]
pipeline = Pipeline(steps) # define the pipeline object.
parameteres = {'SVM__C':[0.001,0.1,10,100,10e5], 'SVM__gamma':[0.1,0.01]}
grid = GridSearchCV(pipeline, param_grid=parameteres, cv=5)
grid.fit(X_train, y_train)
```

- Spark ML Pipelines (Similar to SkLearn)

```
tokenizer = Tokenizer(inputCol="text", outputCol="words")
hashingTF = HashingTF(inputCol=tokenizer.getOutputCol(), outputCol="features")
lr = LogisticRegression(maxIter=10, regParam=0.001)
pipeline = Pipeline(stages=[tokenizer, hashingTF, lr])
```

SystemML (VLDB'16)

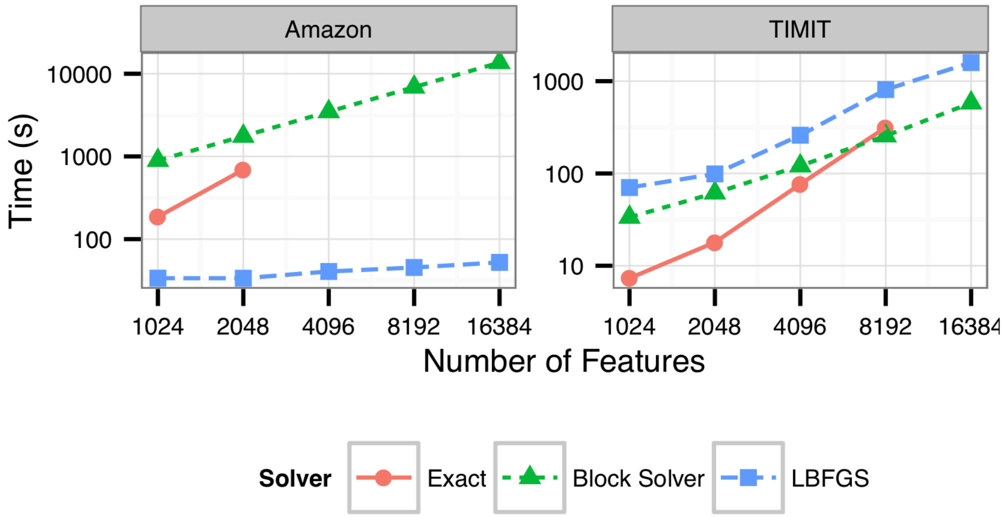
- Developed at IBM
- **Domain specific language** for describing ML algorithms
  - Python/R like but **not embedded**
  - Optimizer and runtime to execute on **Apache Spark**
- Explored range of optimizations
  - Data repartitioning
  - Caching
  - Distributed matrix representations

```
X = read($inFile);
r = $rank; lambda = $lambda;
U = rand(rows=nrow(X), cols=r, min=-1.0, max=1.0);
V = rand(rows=r, cols=ncol(X), min=-1.0, max=1.0);
W = (X != 0);
mi = $maxiter; mii = r; i = 0; is_U = TRUE;
while(i < mi) {
  i = i + 1; ii = 1;
  if (is_U)
     G = (W * (U %*% V - X)) %*% t(V) + lambda * U;
  else
     G = t(U) %*% (W * (U %*% V - X)) + lambda * V;
  norm_G2 = sum(G ^ 2); norm_R2 = norm_G2;
  R = -G; S = R;
  while(norm_R2 > 10E-9 * norm_G2 & ii <= mii) {
    if (is_U) {
      HS = (W * (S %*% V)) %*% t(V) + lambda * S;
      alpha = norm_R2 / sum (S * HS);
      U = U + alpha * S;
    } else {
      HS = t(U) %*% (W * (U %*% S)) + lambda * S;
      alpha = norm_R2 / sum (S * HS);
      V = V + alpha * S;
    }
    R = R - alpha * HS;
    old_norm_R2 = norm_R2; norm_R2 = sum(R ^ 2);
    S = R + (norm_R2 / old_norm_R2) * S;
    ii = ii + 1;
  }
  is_U = ! is_U;
}
write(U, $outUFile, format = "text");
write(V, $outVFile, format = "text");
```

Keystone ML (ICDE'17)

- Developed in AMPLab@Berkeley
- Pipelines of **ML algorithms** and optimization on top of Spark
  - **Embedded Scala DSL**
  - Outperformed SystemML
- Cost based optimize to select best version of learning algorithm based on inputs
  - Example: QR vs L-BFGS

```
val textClassifier = Trim andThen
    LowerCase andThen
    Tokenizer andThen
    NGramsFeaturizer(1 to 2) andThen
    TermFrequency(x => 1) andThen
    (CommonSparseFeatures(1e5), data) andThen
    (LinearSolver(), data, labels)
val predictions = textClassifier(testData)
```



Languages vs Algorithm Libraries

- Increased focus on deep learning → empirical risk minimization for complex **differentiable models**
- Research shifts from algorithm design to **model design**
- **Deep Learning Frameworks:**Theano (2008), Caffe (2014), MXNet (2015), TensorFlow (2015), PyTorch (2016)
  - Combine **automatic differentiation** with **hardware acceleration**

Review of
Automatic Differentiation

Automatic Differentiation

- Method of computing **numeric derivatives** of a **program** by **tracking** the **forward execution** of that **program**
- Other methods for computing derivatives
  - **Manual implementation:** the standard method in deep learning prior to these frameworks
    - laborious and **error prone**!
  - **Numerical differentiation:** using finite differences
    - Easy, costly and sensitive to numerical precision
  - **Symbolic differentiation:** using computer algebraic systems
    - Expressions can grow exponentially

Illustration from "*Automatic Differentiation in Machine Learning: a Survey*"

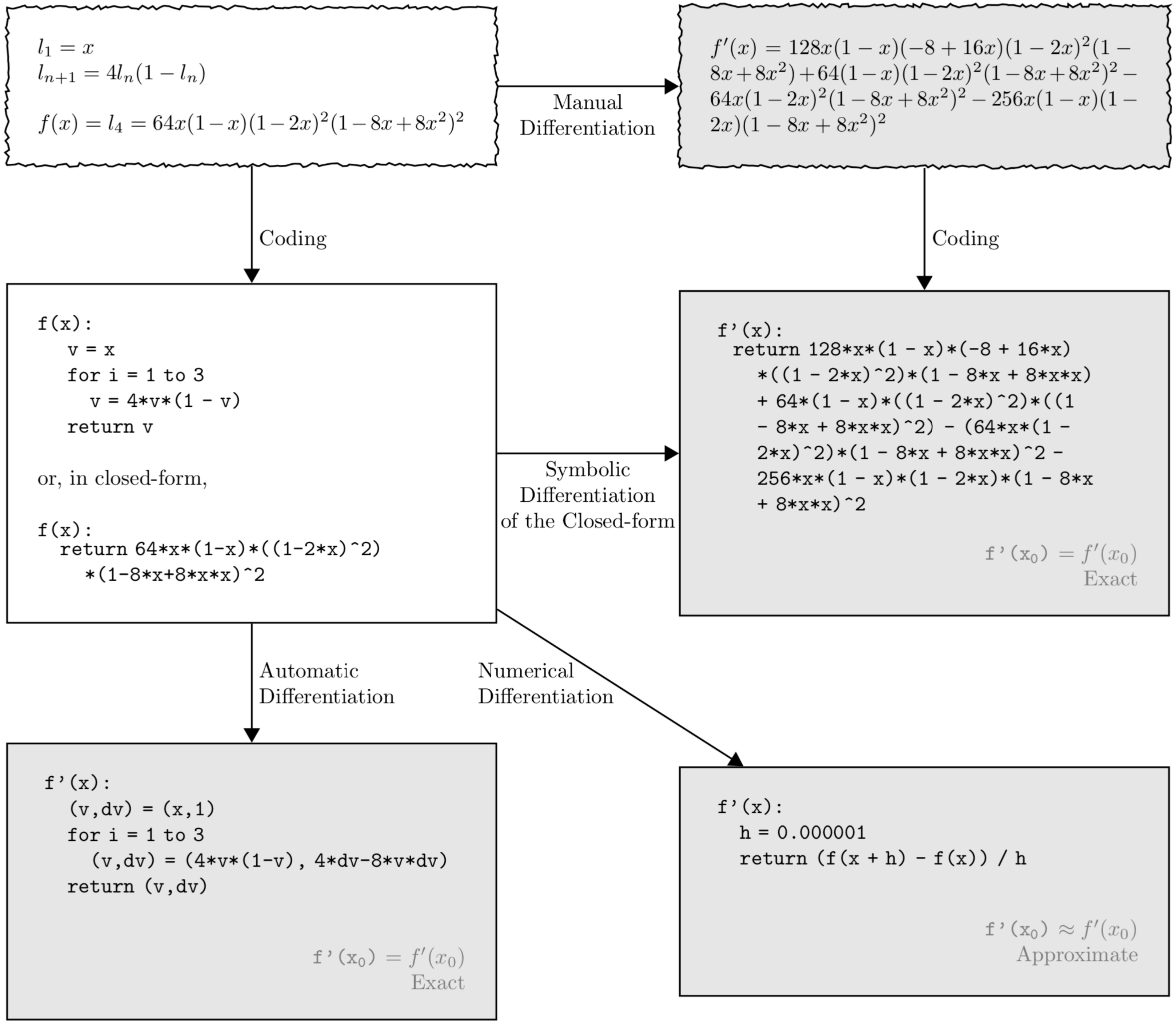
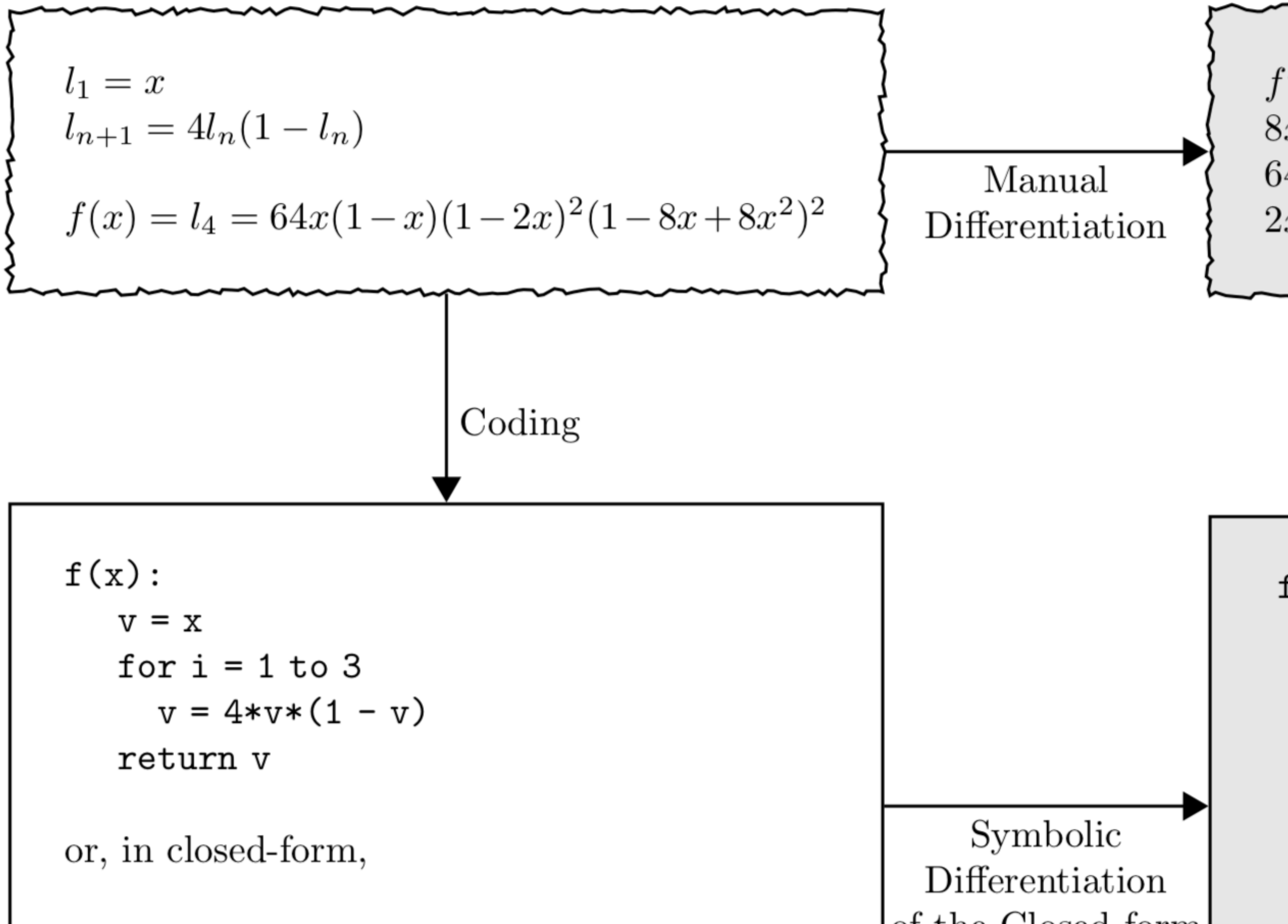
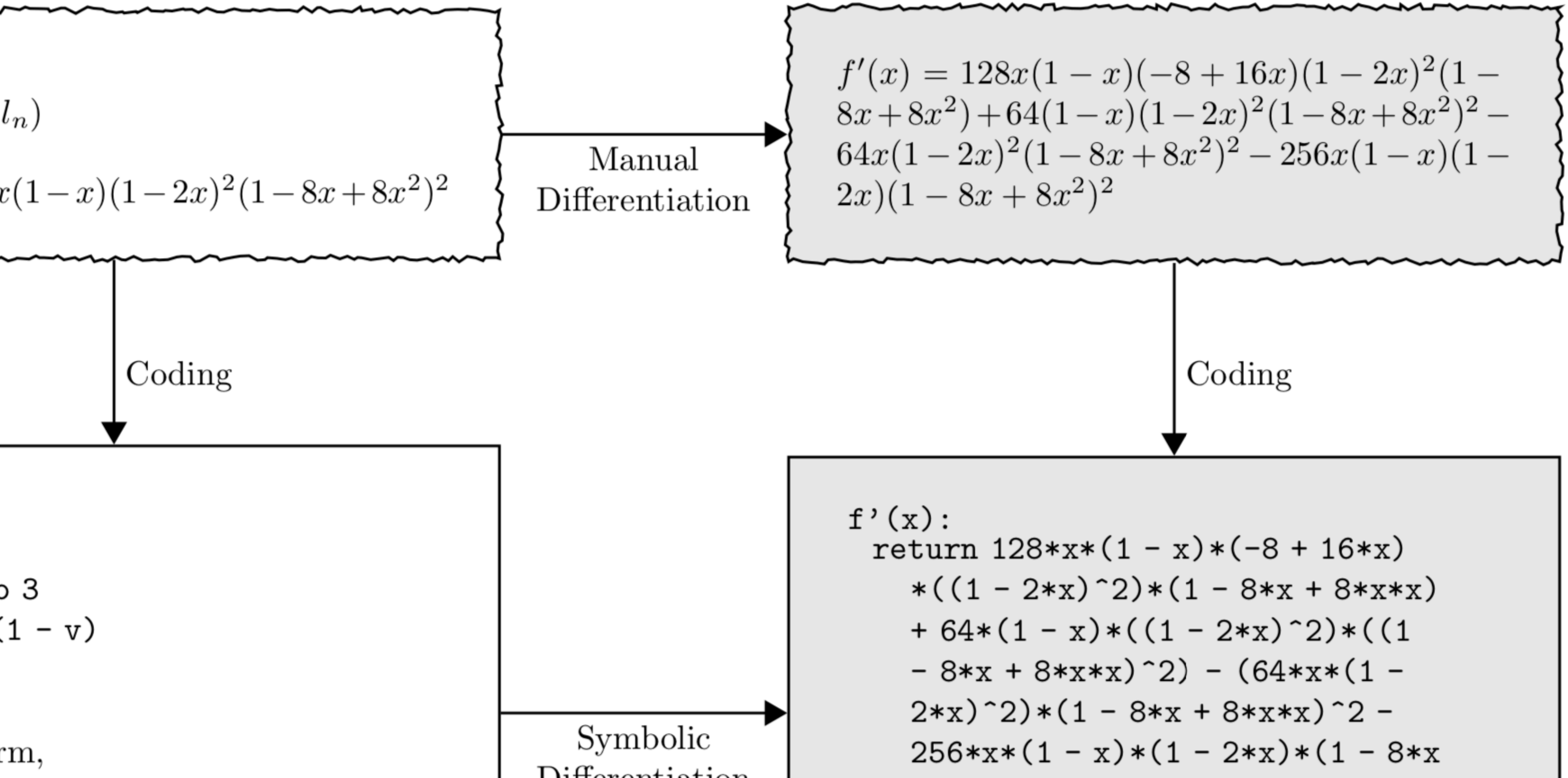


Illustration from "*Automatic Differentiation in Machine Learning: a Survey*"




$l_n)$
$x(1-x)(1-2x)^2(1-8x+8x^2)^2$
Manual
Differentiation
$f'(x) = 128x(1-x)(-8+16x)(1-2x)^2(1-8x+8x^2)+64(1-x)(1-2x)^2(1-8x+8x^2)^2-64x(1-2x)^2(1-8x+8x^2)^2-256x(1-x)(1-2x)(1-8x+8x^2)^2$
Coding
Coding
3
(1 - v)
rm,
Symbolic
f'(x):
return 128*x*(1 - x)*(-8 + 16*x)
*((1 - 2*x)^2)*(1 - 8*x + 8*x*x)
+ 64*(1 - x)*((1 - 2*x)^2)*((1
- 8*x + 8*x*x)^2) - (64*x*(1 -
2*x)^2)*(1 - 8*x + 8*x*x)^2 -
256*x*(1 - x)*(1 - 2*x)*(1 - 8*x

$$l_1 = x$$
$$l_{n+1} = 4l_n(1 - l_n)$$

$$f(x) = l_4 = 64x(1-x)(1-2x)^2(1-8x+8x^2)^2$$

Manual Differentiation

$$f'(x) = 128x(1-x)(-8+16x)(1-2x)^2(1-8x+8x^2)+64(1-x)(1-2x)^2(1-8x+8x^2)^2-64x(1-2x)^2(1-8x+8x^2)^2-256x(1-x)(1-2x)(1-8x+8x^2)^2$$

How I used to do this as a graduate student (2010).

Coding

Coding

```
f(x):
   v = x
   for i = 1 to 3
     v = 4*v*(1 - v)
   return v
```

or, in closed-form,

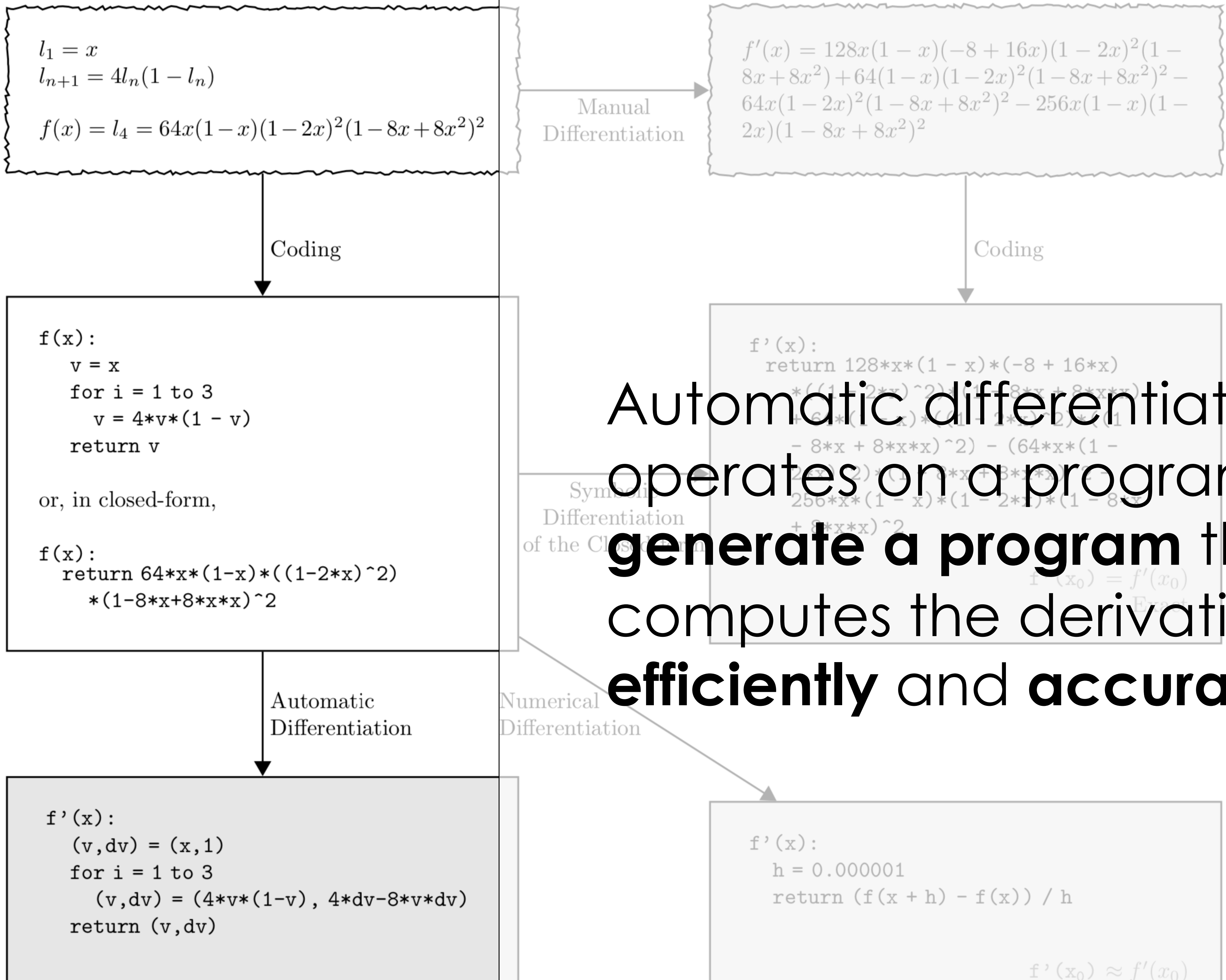
```
f(x):
  return 64*x*(1-x)*((1-2*x)^2)
    *(1-8*x+8*x*x)^2
```

How I would cheat using Mathematica.

Symbolic Differentiation of the Closed-form

```
f'(x):
 return 128*x*(1 - x)*(-8 + 16*x)
   *((1 - 2*x)^2)*(1 - 8*x + 8*x*x)
   + 64*(1 - x)*((1 - 2*x)^2)*((1
   - 8*x + 8*x*x)^2) - (64*x*(1 -
   2*x)^2)*(1 - 8*x + 8*x*x)^2 -
   256*x*(1 - x)*(1 - 2*x)*(1 - 8*x
   + 8*x*x)^2
```

f'(x_0) $= f'(x_0)$
Exact



Automatic differentiation operates on a program to **generate a program** that computes the derivative **efficiently** and **accurately**.

Key Ideas in Automatic Differentiation

- Leverage **Chain Rule** to reason about function composition

$$\frac{\partial}{\partial x} f\left(g\left(x\right)\right) = \dot{f}\left(g\left(x\right)\right) \frac{\partial}{\partial x} g\left(x\right)$$

- Two modes of automatic differentiation
  - **Forward differentiation**: computes derivative during execution
    - efficient for single derivative with multiple outputs
  - **Backward differentiation (back-propagation):** computes derivative (gradient) by reverse evaluation of the computation graph
    - Efficient for multiple derivative (gradient) calculation + **Requires caching**

Forward Differentiation (Example)

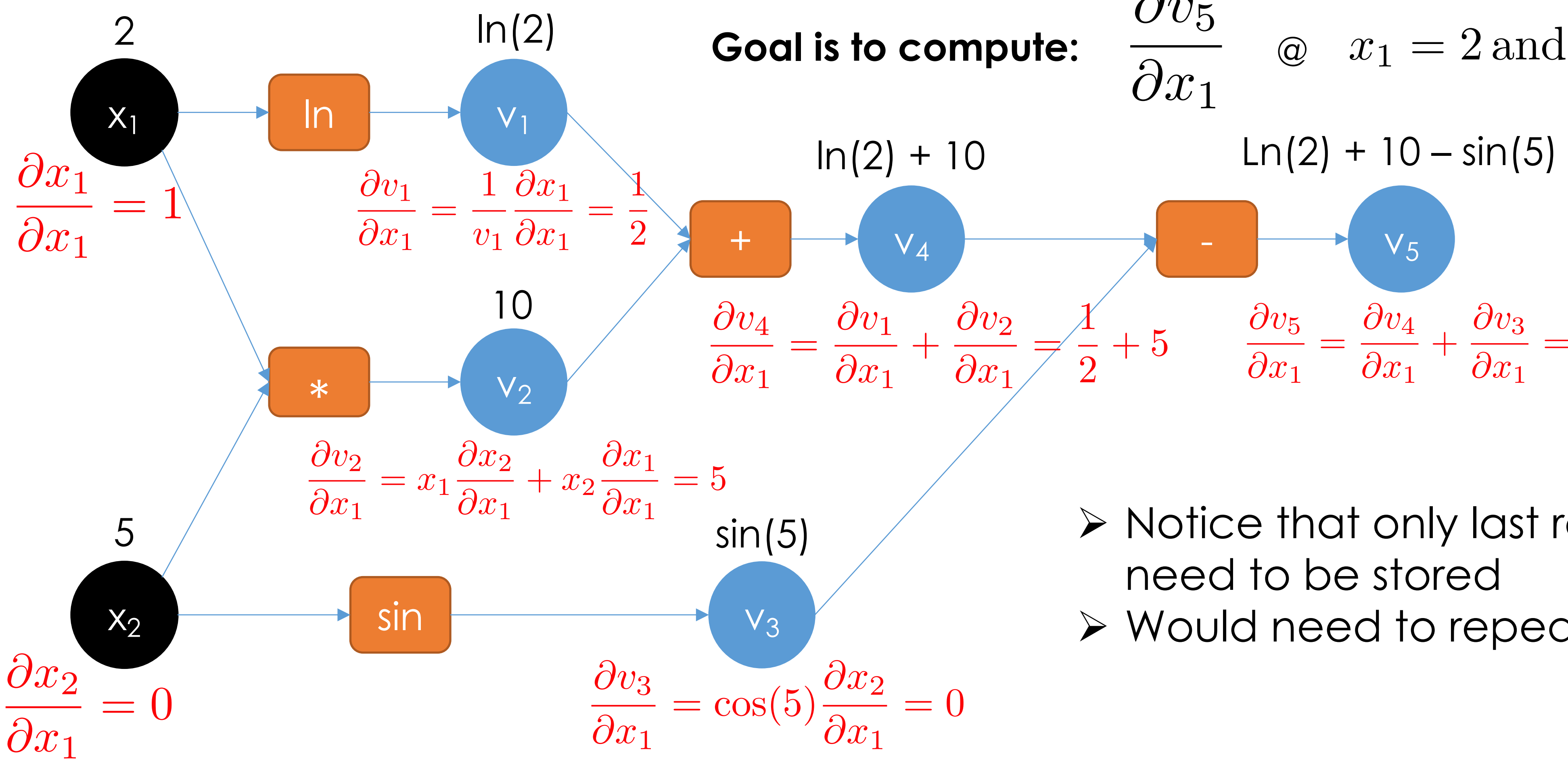
$$f\left(x_1, x_2\right) = \ln\left(x_1\right) + x_1 x_2 - \sin\left(x_2\right)$$

Goal is to compute: $\dfrac{\partial v_5}{\partial x_1}$ @ $x_1 = 2 \,\text{and}\, x_2 = 5$

Forward Differentiation (Example)

$$f\left(x_1, x_2\right) = \ln\left(x_1\right) + x_1 x_2 - \sin\left(x_2\right)$$

Goal is to compute: $\frac{\partial v_5}{\partial x_1}$ @ $x_1 = 2 \text{ and } x_2 = 5$

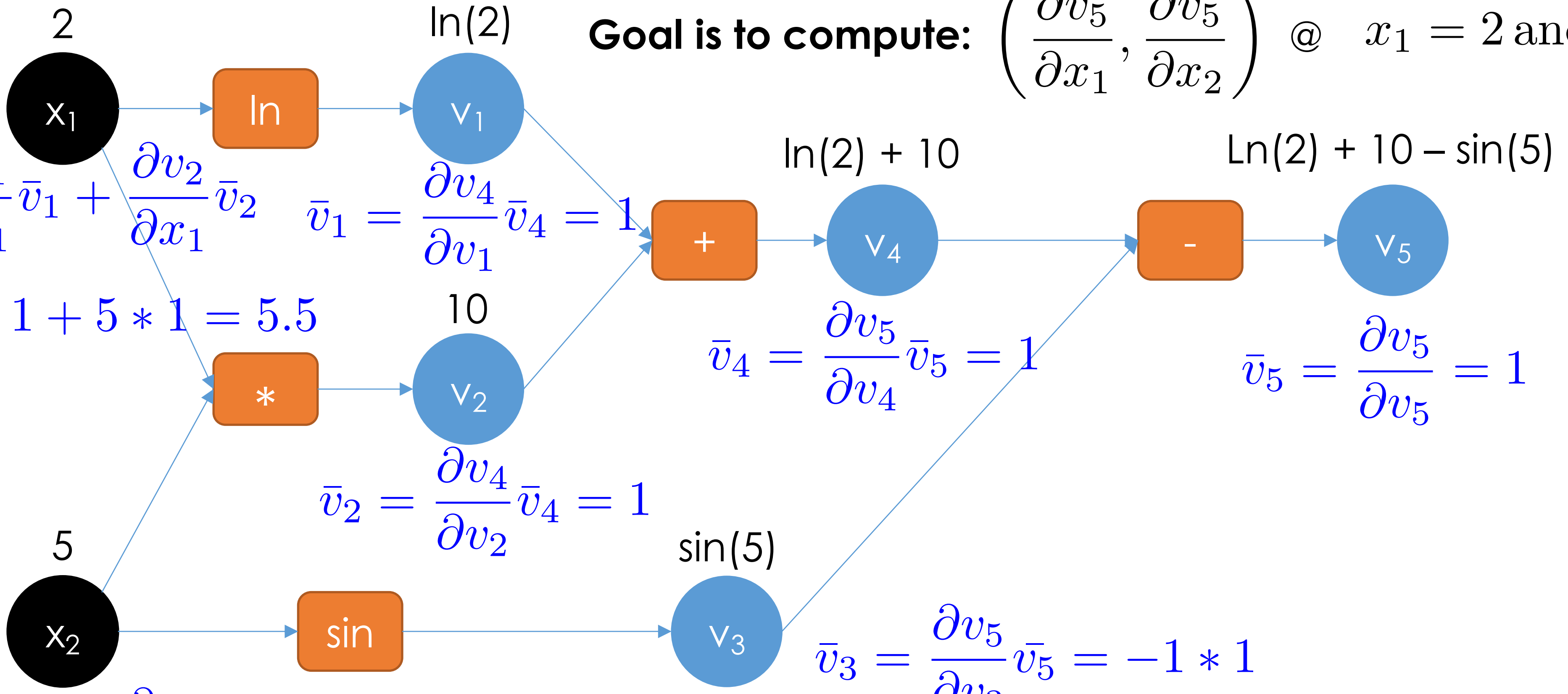


- Notice that only last results need to be stored
- Would need to repeat for x_2

Backward (Reverse) Differentiation

$$f(x_1, x_2) = \ln(x_1) + x_1 x_2 - \sin(x_2)$$

Goal is to compute: $\left(\frac{\partial v_5}{\partial x_1}, \frac{\partial v_5}{\partial x_2}\right)$ @ $x_1 = 2 \text{ and } x_2 = 5$



$$\bar{v}_5 = \frac{\partial v_5}{\partial v_5} = 1$$

$$\bar{v}_4 = \frac{\partial v_5}{\partial v_4}\bar{v}_5 = 1$$

$$\bar{v}_3 = \frac{\partial v_5}{\partial v_3}\bar{v}_5 = -1 * 1$$

$$\bar{v}_1 = \frac{\partial v_4}{\partial v_1}\bar{v}_4 = 1$$

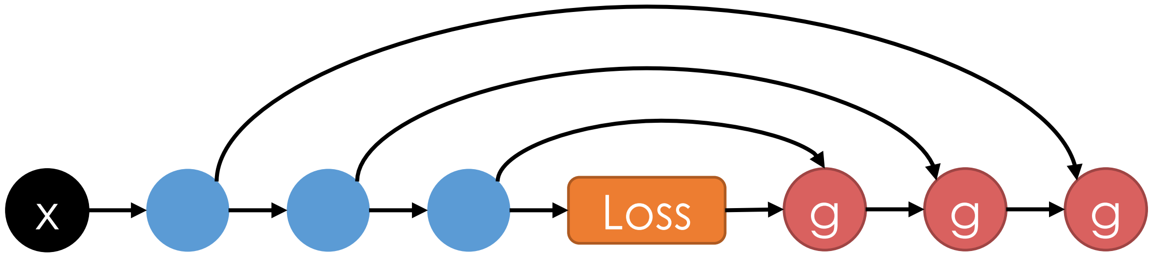
$$\bar{v}_2 = \frac{\partial v_4}{\partial v_2}\bar{v}_4 = 1$$

$$\bar{x}_1 = \frac{\partial v_1}{\partial x_1}\bar{v}_1 + \frac{\partial v_2}{\partial x_1}\bar{v}_2 = \frac{1}{2} * 1 + 5 * 1 = 5.5$$

$$\bar{x}_2 = \frac{\partial v_2}{\partial x_2}\bar{v}_2 + \frac{\partial v_3}{\partial x_2}\bar{v}_3 = 2 * 1 + \cos(5) * -1 \approx 1.716$$

Backward (Reverse) Differentiation

- Performs well when **computing large gradients** relative to number of function outputs
  - When might forward differentiation perform well? Why?
- Requires **caching** or **recomputing** intermediate activations from forward pass
  - Active research on what to recompute vs cache



Deep Learning Frameworks

Declarative vs Imperative Abstractions

- **Declarative** *(define-and-run)*: Embedded DSL used to construct **static computation graph**
  - Examples: Theano (2010), Caffe (2014), TensorFlow (2015)
  - **Easier** to optimize, distribute, and export models
- **Imperative** *(define-by-run)*: Embedded DSL used to directly compute output resulting in a **dynamic computation graph** defined by the program
  - Examples: Chainer (2015), autograd (2016), PyTorch (2017)
  - **Interpreted execution** of inference and gradient
  - **Easier** to program and debug
- **Hybrid Approaches**: Current research
  - TensorFlow Eager, MXNet

Theano – Original Deep Learning Framework

- First developed at the **University of Montreal** (2008)
  - from **Yoshua Bengio's** group
- **Abstraction:** Python embedded DSL (as a library) to construct symbolic expression graphs for complex mathematical expressions
- **System:** a **compiler** for **mathematical expressions** in Python
  - Optimizes mathematical expressions (e.g., *(A+b)(A+b)=(A+b)^2*)
  - CPU/**GPU acceleration**
  - Also … **automatic differentiation**

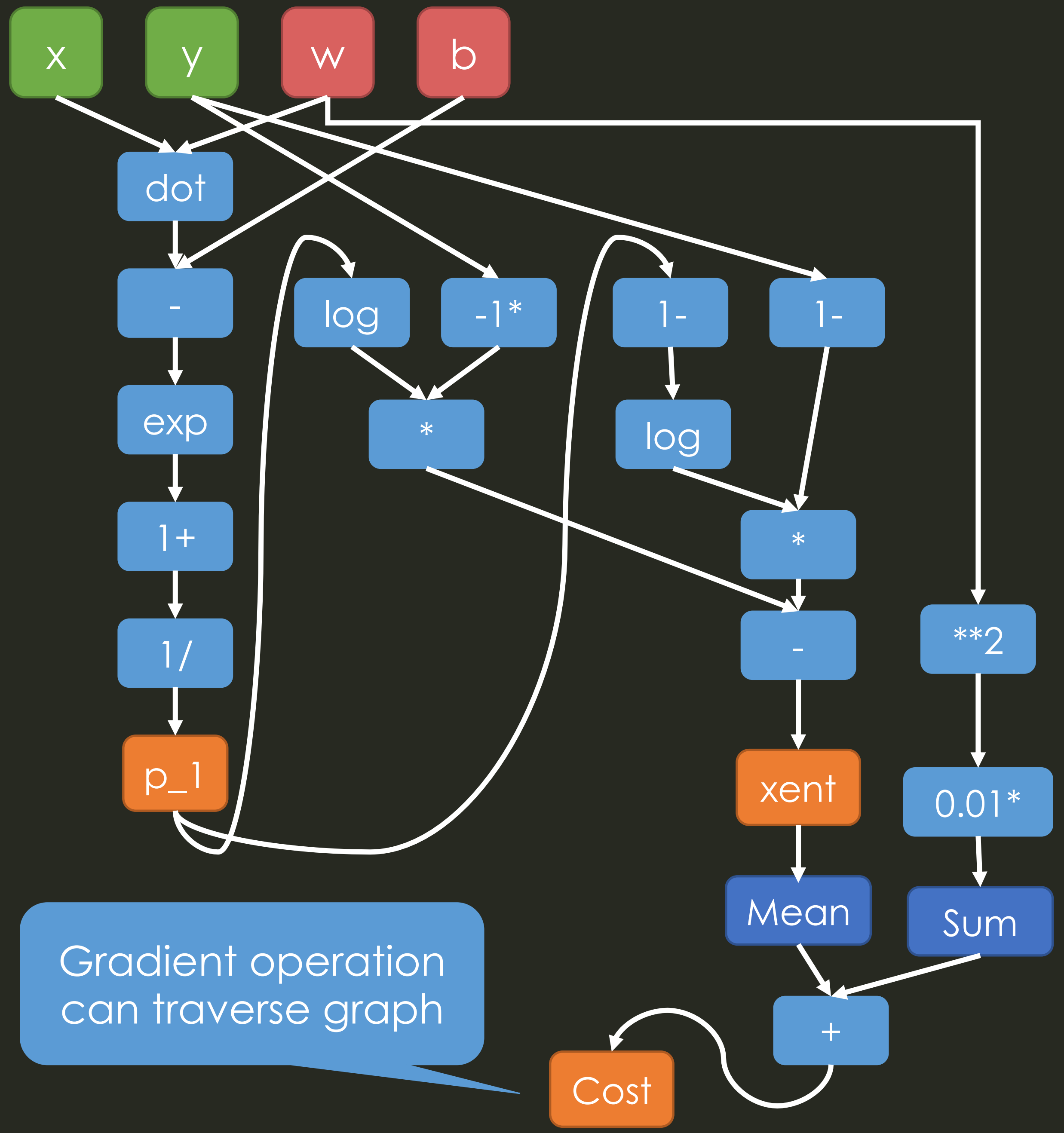
```python
import numpy
import theano.tensor as T
from theano import shared, function

x = T.matrix()
y = T.lvector()
w = shared(numpy.random.randn(100))
b = shared(numpy.zeros(()))

print "Initial model:"
print w.get_value(), b.get_value()

p_1 = 1 / (1 + T.exp(-T.dot(x, w)-b))
xent = -y*T.log(p_1) - (1-y)*T.log(1-p_1)
cost = xent.mean() + 0.01*(w**2).sum()
gw,gb = T.grad(cost, [w,b])
prediction = p_1 > 0.5
```

Declaring Variables

Building Expression Graph
Note that this looks like a NumPy expression

What is the value (type) of prediction?

This is more difficult to debug and reason about.



Gradient operation can traverse graph

```python
import numpy
import theano.tensor as T
from theano import shared, function

x = T.matrix()
y = T.lvector()
w = shared(numpy.random.randn(100))
b = shared(numpy.zeros(()))

print "Initial model:"
print w.get_value(), b.get_value()

p_1 = 1 / (1 + T.exp(-T.dot(x, w)-b))
xent = -y*T.log(p_1) - (1-y)*T.log(1-p_1)
cost = xent.mean() + 0.01*(w**2).sum()
gw,gb = T.grad(cost, [w,b])
prediction = p_1 > 0.5
```

Declaring Variables

Building Expression Graph
Note that this looks like a NumPy expression

What is the value (type) of prediction?

This is more difficult to debug and reason about.

```python
predict = function(inputs=[x],
                   outputs=prediction)
train = function(
    inputs=[x,y],
    outputs=[prediction, xent],
    updates={w: w - 0.1*gw, b: b - 0.1*gb})

N = 4
feats = 100
D = (numpy.random.randn(N, feats),
numpy.random.randint(size=N,low=0, high=2))
training_steps = 10
for i in range(training_steps):
    pred, err = train(D[0], D[1])
print "Final model:",
print w.get_value(), b.get_value()
print "target values for D", D[1]
print "prediction on D", predict(D[0])
```

Updates shared variables after computation

Instantiating Values

Function call compiles graphs into optimized native execution.

Theano Compilation of Functions

- **Rewriting** (simplify) mathematical expression
  - Exp(log(x)) = x
- Duplicate **code elimination**
  - Important because gradient rewrites introduce redundancy
    - Recall gradient calculations extend graph via the chain rule

Theano Compilation of Functions

Addresses **numerical stability** of operations

- Example: for x = 709, x = 710 what is the value of

$$\log(1 + \exp(x)) =$$

  - for x = 709 ➔ 709
  - for x = 710 ➔ inf
  - Rewritten as x for x > 709

Theano Compilation of Functions

- Rewrite subgraphs to more efficient forms
  - pow(x,2) → square(x)
  - Tensor slicing → memory aliasing
  - Mapping to best version of GEMM routines

Theano Compilation of Functions

- GPU versions of ops are introduced (where possible)
- Copy routines are added to move data

Theano Compilation of Functions

- Generate and link C++ and CUDA implementations of operators
  - Picking from existing implementations
  - Specialization for different dtypes

What happened to Theano?

- Fairly advanced compared to TensorFlow (TF) in 2016
  - Symbolic gradient optimization and wide range of operators
  - Initially **faster than TensorFlow**
- What happened?
  - Didn't have the backing of a large industrial group
    - TensorFlow was being pushed heavily by Google
  - Did not support multi-GPU/distributed computation and limited support for user defined parallelization
  - TensorFlow had more built-in deep learning operators
  - Theano lacked visualization tools (e.g., TensorBoard)
  - Complaints about error messages…?

PyTorch

- **Imperative DL library** which works like NumPy (on GPUs)

```
if torch.cuda.is_available():
device = torch.device("cuda")          # a CUDA device object
y = torch.ones_like(x, device=device)  # directly create a tensor on GPU
x = x.to(device)                       # or just use strings ``.to("cuda")``
z = x + y
print(z)
print(z.to("cpu", torch.double))       # er!
```

tensor([2.0814], device='cuda:0')

tensor([2.0814], dtype=torch.float64)

- and supports automatic differentiation

```
x = torch.ones(2, 2, requires_grad=True)
y = x + 2
print(y)
z = y * y * 3
out = z.mean()
print(out)
out.backward()
print(x.grad)
```

tensor([[3., 3.],
[3., 3.]], grad_fn=<AddBackward0>)

tensor(27., grad_fn=<MeanBackward0>)

tensor([[4.5000, 4.5000],
[4.5000, 4.5000]])

This weeks readings

Reading for the Week

- Automatic differentiation in ML: Where we are and where we should be going
  - NeurIPS'18
  - Provides an overview of the state of automatic differentiation
- TensorFlow: A System for Large-Scale Machine Learning
  - OSDI'16
  - The primary TensorFlow paper discusses system and design goals
- JANUS: Fast and Flexible Deep Learning via Symbolic Graph Execution of Imperative Programs
  - NSDI'19
  - Recent work exploring a method to bridge Declarative and Imperative approaches in TensorFlow

Extra Suggested Reading

- Automatic Differentiation in Machine Learning: a Survey(JMLR'18)
  - Longer discussion on automatic differentiation in ML
- Theano: A CPU and GPU Math Compiler in Python (SciPy'10)
  - Great overview of AD and Theano system
- TensorFlow Eager: A Multi-Stage, Python-Embedded DSL for Machine Learning (arXiv'19)
  - Good follow-up to TF paper addressing limitations

Automatic differentiation in ML: Where we are and where we should be going?

Bart van Merriënboer, Olivier Breuleux, Arnaud Bergeron, Pascal Lamblin

From Mila (home of Theano) and Google Brain (home of TF)

Automatic differentiation in ML: Where we are and where we should be going?

- **Context:** A **vision paper** that outlines the **current state of automatic differentiation** techniques and proposes a new functional, typed **intermediate representation** (IR)

- **Key Idea:** Observe convergence of imperative and declarative approaches and draws connections to compilers → argues for the need for a common IR like those found in modern compilers.

- **Contribution:** Frames problem space and range of techniques.

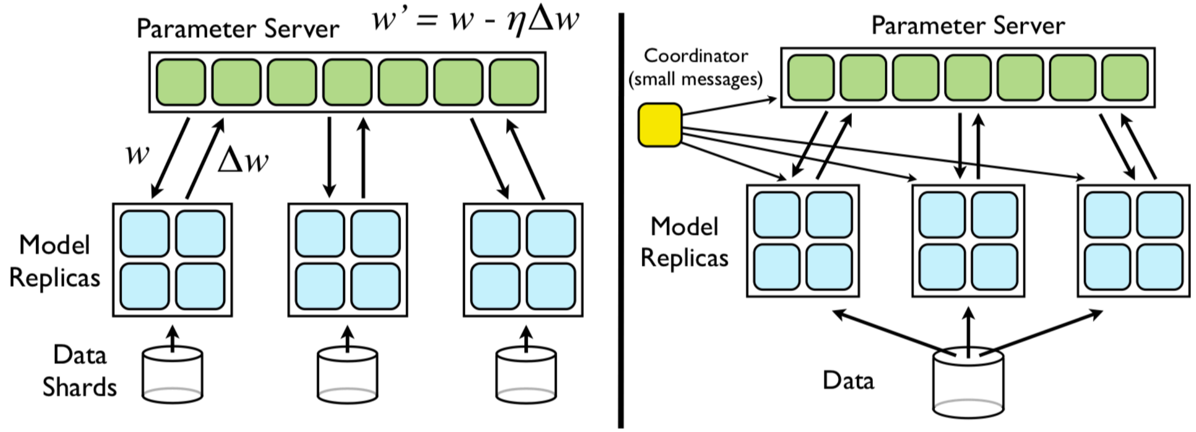
- **Rational for Reading:** condensed context and some insights for future research directions

TensorFlow: A System for Large-Scale Machine Learning

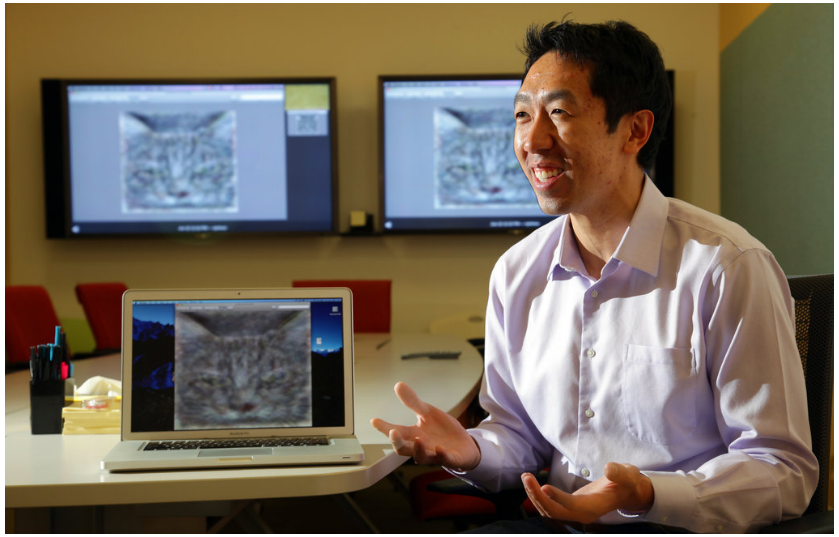
Large fraction of Google Brain team under Jeff Dean

Context

- Need for distributed training for Deep Learning
- Parameter server abstractions were too general
  - Difficult to use



DistBelief Framework

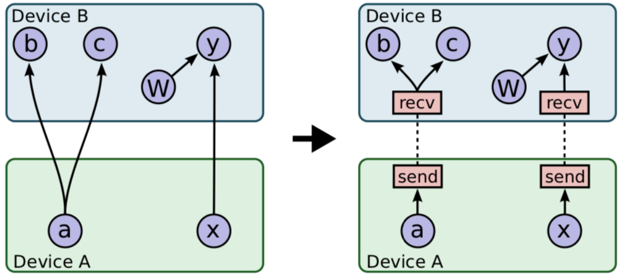


- Theano not designed for distributed setting

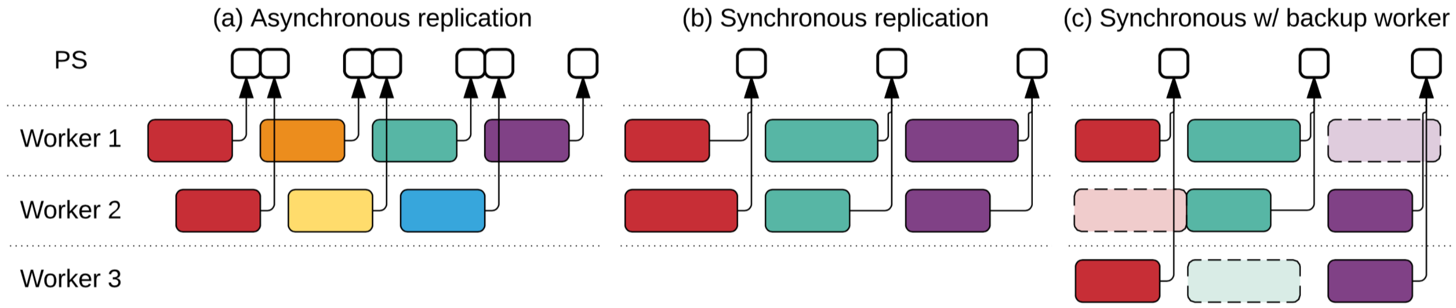
Big Ideas

- Adopts a **dataflow programming** abstraction
  - Inspired by distributed **data processing systems** (@ google)
  - Resulting abstraction is very **similar to Theano**
- Fine grained placement of operations on devices

```
c = []
for d in ['/device:GPU:2', '/device:GPU:3']:
  with tf.device(d):
    a = tf.constant([1.0, 2.0, 3.0, 4.0, 5.0, 6.0], shape=[2, 3])
    b = tf.constant([1.0, 2.0, 3.0, 4.0, 5.0, 6.0], shape=[3, 2])
    c.append(tf.matmul(a, b))
```



- Support multiple distributed concurrency protocols



Recent advances in TensorFlow

➢ **Keras** : high-level layer composition API

```
# Define the model sequentially
model = tf.keras.Sequential([
    # Adds two densely-connected layers with 64 units:
    layers.Dense(64, activation='relu', input_shape=(32,)),
    layers.Dense(64, activation='relu'),
    # Add a softmax layer with 10 output units:
    layers.Dense(10, activation='softmax')])

# Setup the model and training routines
model.compile(optimizer=tf.train.AdamOptimizer(0.001),
              loss='categorical_crossentropy',
              metrics=['accuracy'])

# Invent some Data
data = np.random.random((1000, 32))
labels = random_one_hot_labels((1000, 10))
# Train the model
model.fit(data, labels, epochs=10, batch_size=32)
# Make predictions
result = model.predict(data, batch_size=32)
```

Discussion on TensorFlow Eager in next section.

What to think about when reading

- Relationship and comparisons to Theano?
- Support for distributed computing and exposed abstraction?
- What are the implications of design decisions on an Eager Execution

Additional Reading

- [TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems]

JANUS: Fast and Flexible Deep Learning via Symbolic Graph Execution of Imperative Programs

Eunji Jeong* et al. at Seoul National University

*Currently visiting in the RISE Lab

Context

- In response to PyTorch Google recently released **TesorFlow Eager**

```
x = tf.Variable(tf.ones([2,2]))
with tf.GradientTape() as tape:
  y = x + 2
  z = y * y * 3
  out = tf.math.reduce_mean(z)
  print(out)
grad = tape.gradient(out, x)
print(grad)
```

  - **Pro:** Simplifies programming especially for **dynamic graphs**
  - **Con: limited opt.** and **interpreted execution** → degraded training performance

Big Ideas

- Convert imperative executions into dataflow graphs
- **Combines:**
  - Python **program analysis** to generate symbolic graph
  - Execution **profiling** to observe outcomes of dynamic comp.
- Leverage profiling to **speculate** on dynamics when constructing symbolic graph
  - **Check assumptions** at runtime
- If **assumptions are violated** *safely* re-execute imperative code

What to think about when reading

- Opportunities for re-execution during training process
- Additional optimizations that could be introduced
- Implications on model deployment (inference)

Suggested Additional Reading

- TensorFlow Eager: A Multi-Stage, Python-Embedded DSL for Machine Learning (SysML'19)

Done!