SVM multiclass classification in 10 steps

```
import numpy as np

# load digits dataset
from sklearn import datasets
digits = datasets.load_digits()

# define training set size
n_samples = len(digits.images)
n_training = int( 0.9 * n_samples )

data = np.ascontiguousarray( digits.data, dtype=np.double )
labels = np.ascontiguousarray( digits.target.reshape(n_samples, 1),
                               dtype=np.double )

from daal.data_management import HomogenNumericTable

train_data = HomogenNumericTable( data[:n_training] )
train_labels = HomogenNumericTable( labels[:n_training] )

test_data = HomogenNumericTable( data[n_training:] )
```

SVM multiclass classification in 10 steps

```
import numpy as np

# load digits dataset
from sklearn import datasets
digits = datasets.load_digits()

# define training set size
n_samples = len(digits.images)
n_training = int( 0.9 * n_samples )

data = np.ascontiguousarray( digits.data, dtype=np.double )
labels = np.ascontiguousarray( digits.target.reshape(n_samples, 1),
                               dtype=np.double )

from daal.data_management import HomogenNumericTable

train_data = HomogenNumericTable( data[:n_training] )
train_labels = HomogenNumericTable( labels[:n_training] )

test_data = HomogenNumericTable( data[n_training:] )
```

1. enjoy sklearn datasets import module

SVM multiclass classification in 10 steps

```
import numpy as np

# load digits dataset
from sklearn import datasets
digits = datasets.load_digits()

# define training set size
n_samples = len(digits.images)
n_training = int( 0.9 * n_samples )

data = np.ascontiguousarray( digits.data, dtype=np.double )
labels = np.ascontiguousarray( digits.target.reshape(n_samples, 1),
                               dtype=np.double )

from daal.data_management import HomogenNumericTable

train_data = HomogenNumericTable( data[:n_training] )
train_labels = HomogenNumericTable( labels[:n_training] )

test_data = HomogenNumericTable( data[n_training:] )
```

1. enjoy sklearn datasets import module
2. require a contiguous array

SVM multiclass classification in 10 steps

```
import numpy as np

# load digits dataset
from sklearn import datasets
digits = datasets.load_digits()

# define training set size
n_samples = len(digits.images)
n_training = int( 0.9 * n_samples )

data = np.ascontiguousarray( digits.data, dtype=np.double )
labels = np.ascontiguousarray( digits.target.reshape(n_samples, 1),
                               dtype=np.double )

from daal.data_management import HomogenNumericTable

train_data = HomogenNumericTable( data[:n_training] )
train_labels = HomogenNumericTable( labels[:n_training] )

test_data = HomogenNumericTable( data[n_training:] )
```

1. enjoy sklearn datasets import module
2. require a contiguous array
3. create instances of HomogenNumericTable

training algorithm setup

```
from daal.algorithms.svm import training as svm_training
from daal.algorithms.svm import prediction as svm_prediction
from daal.algorithms.multi_class_classifier import training as
    multiclass_training
from daal.algorithms.classifier import training as training_params

kernel = rbf.Batch_Float64DefaultDense()
kernel.parameter.sigma = 0.001

# Create two class svm classifier
# training alg
twoclass_train_alg = svm_training.Batch_Float64DefaultDense()
twoclass_train_alg.parameter.kernel = kernel
twoclass_train_alg.parameter.C = 1.0
# prediction alg
twoclass_predict_alg = svm_prediction.Batch_Float64DefaultDense()
twoclass_predict_alg.parameter.kernel = kernel

# Create a multiclass classifier object (training)
train_alg = multiclass_training.Batch_Float64OneAgainstOne()
train_alg.parameter.nClasses = 10
train_alg.parameter.training = twoclass_train_alg
train_alg.parameter.prediction = twoclass_predict_alg
```

training algorithm setup

```
from daal.algorithms.svm import training as svm_training
from daal.algorithms.svm import prediction as svm_prediction
from daal.algorithms.multi_class_classifier import training as
    multiclass_training
from daal.algorithms.classifier import training as training_params

kernel = rbf.Batch_Float64DefaultDense()
kernel.parameter.sigma = 0.001

# Create two class svm classifier
# training alg
twoclass_train_alg = svm_training.Batch_Float64DefaultDense()
twoclass_train_alg.parameter.kernel = kernel
twoclass_train_alg.parameter.C = 1.0
# prediction alg
twoclass_predict_alg = svm_prediction.Batch_Float64DefaultDense()
twoclass_predict_alg.parameter.kernel = kernel

# Create a multiclass classifier object (training)
train_alg = multiclass_training.Batch_Float64OneAgainstOne()
train_alg.parameter.nClasses = 10
train_alg.parameter.training = twoclass_train_alg
train_alg.parameter.prediction = twoclass_predict_alg
```

4. define kernel and kernel parameters

training algorithm setup

```
from daal.algorithms.svm import training as svm_training
from daal.algorithms.svm import prediction as svm_prediction
from daal.algorithms.multi_class_classifier import training as
    multiclass_training
from daal.algorithms.classifier import training as training_params

kernel = rbf.Batch_Float64DefaultDense()
kernel.parameter.sigma = 0.001

# Create two class svm classifier
# training alg
twoclass_train_alg = svm_training.Batch_Float64DefaultDense()
twoclass_train_alg.parameter.kernel = kernel
twoclass_train_alg.parameter.C = 1.0
# prediction alg
twoclass_predict_alg = svm_prediction.Batch_Float64DefaultDense()
twoclass_predict_alg.parameter.kernel = kernel

# Create a multiclass classifier object (training)
train_alg = multiclass_training.Batch_Float64OneAgainstOne()
train_alg.parameter.nClasses = 10
train_alg.parameter.training = twoclass_train_alg
train_alg.parameter.prediction = twoclass_predict_alg
```

4. define kernel and kernel parameters
5. create two class svm classifier

training algorithm setup

```
from daal.algorithms.svm import training as svm_training
from daal.algorithms.svm import prediction as svm_prediction
from daal.algorithms.multi_class_classifier import training as
    multiclass_training
from daal.algorithms.classifier import training as training_params

kernel = rbf.Batch_Float64DefaultDense()
kernel.parameter.sigma = 0.001

# Create two class svm classifier
# training alg
twoclass_train_alg = svm_training.Batch_Float64DefaultDense()
twoclass_train_alg.parameter.kernel = kernel
twoclass_train_alg.parameter.C = 1.0
# prediction alg
twoclass_predict_alg = svm_prediction.Batch_Float64DefaultDense()
twoclass_predict_alg.parameter.kernel = kernel

# Create a multiclass classifier object (training)
train_alg = multiclass_training.Batch_Float64OneAgainstOne()
train_alg.parameter.nClasses = 10
train_alg.parameter.training = twoclass_train_alg
train_alg.parameter.prediction = twoclass_predict_alg
```

4. define kernel and kernel parameters
5. create two class svm classifier
6. create multi class svm classifier (training)

training phase

```
# Pass training data and labels
train_alg.input.set(training_params.data, train_data)
train_alg.input.set(training_params.labels, train_labels)

# training
model = train_alg.compute().get(training_params.model)
```

training phase

```
# Pass training data and labels
train_alg.input.set(training_params.data, train_data)
train_alg.input.set(training_params.labels, train_labels)

# training
model = train_alg.compute().get(training_params.model)
```

7. set input data and labels

training phase

```
# Pass training data and labels
train_alg.input.set(training_params.data, train_data)
train_alg.input.set(training_params.labels, train_labels)

# training
model = train_alg.compute().get(training_params.model)
```

7. set input data and labels
8. start training and get model

prediction algorithm setup

```
from daal.algorithms.multi_class_classifier import prediction as
     multiclass_prediction
from daal.algorithms.classifier import prediction as
     prediction_params

# Create a multiclass classifier object (prediction)
predict_alg = multiclass_prediction.
     Batch_Float64DefaultDenseOneAgainstOne()
predict_alg.parameter.nClasses = 10
predict_alg.parameter.training = twoclass_train_alg
predict_alg.parameter.prediction = twoclass_predict_alg
```

prediction algorithm setup

```
from daal.algorithms.multi_class_classifier import prediction as
     multiclass_prediction
from daal.algorithms.classifier import prediction as
     prediction_params

# Create a multiclass classifier object (prediction)
predict_alg = multiclass_prediction.
     Batch_Float64DefaultDenseOneAgainstOne()
predict_alg.parameter.nClasses = 10
predict_alg.parameter.training = twoclass_train_alg
predict_alg.parameter.prediction = twoclass_predict_alg
```

9. create multi class svm classifier (prediction)

prediction phase

```
# Pass a model and input data
predict_alg.input.setModel(prediction_params.model, model)
predict_alg.input.setTable(prediction_params.data, test_data)

# Compute and return prediction results
results = predict_alg.compute().get(prediction_params.prediction)
```

prediction phase

```
# Pass a model and input data
predict_alg.input.setModel(prediction_params.model, model)
predict_alg.input.setTable(prediction_params.data, test_data)

# Compute and return prediction results
results = predict_alg.compute().get(prediction_params.prediction)
```

10. set input model and data

prediction phase

```
# Pass a model and input data
predict_alg.input.setModel(prediction_params.model, model)
predict_alg.input.setTable(prediction_params.data, test_data)

# Compute and return prediction results
results = predict_alg.compute().get(prediction_params.prediction)
```

10. set input model and data
11. start prediction and get labels

Benchmark results

- Test description: 1797 samples total (90% training set, 10% test set), 64 features per sample
- Platform description: Intel Core i5-6300U CPU @ 2.40GHz

	sklearn	pyDAAL	speedup
training time [s]	0.161	0.018	8.9
test time [s]	0.017	0.004	4.3

Advantages and disadvantages

DAAL in general:

Advantages and disadvantages

DAAL in general:

- ✓ very simple installation/setup

Advantages and disadvantages

DAAL in general:

- ✓ very simple installation/setup
- ✓ wide range of algorithms (both for machine l. and for deep l.)

Advantages and disadvantages

DAAL in general:

- ✓ very simple installation/setup
- ✓ wide range of algorithms (both for machine l. and for deep l.)
- ✓ good support through dedicated intel forum (even for non paid versions)

Advantages and disadvantages

DAAL in general:

- ✓ very simple installation/setup
- ✓ wide range of algorithms (both for machine l. and for deep l.)
- ✓ good support through dedicated intel forum (even for non paid versions)
- ✓ DAAL C++ can be called from R and Matlab (see how-to forum posts)

Advantages and disadvantages

DAAL in general:

- ✓ very simple installation/setup
- ✓ wide range of algorithms (both for machine l. and for deep l.)
- ✓ good support through dedicated intel forum (even for non paid versions)
- ✓ DAAL C++ can be called from R and Matlab (see how-to forum posts)
- ✗ documentation is sometimes not exhaustive

Advantages and disadvantages

DAAL in general:

- ✓ very simple installation/setup
- ✓ wide range of algorithms (both for machine l. and for deep l.)
- ✓ good support through dedicated intel forum (even for non paid versions)
- ✓ DAAL C++ can be called from R and Matlab (see how-to forum posts)
- ✗ documentation is sometimes not exhaustive
- ✗ examples cover very simple application cases

Advantages and disadvantages

DAAL in general:

- ✓ very simple installation/setup
- ✓ wide range of algorithms (both for machine l. and for deep l.)
- ✓ good support through dedicated intel forum (even for non paid versions)
- ✓ DAAL C++ can be called from R and Matlab (see how-to forum posts)
- ✗ documentation is sometimes not exhaustive
- ✗ examples cover very simple application cases

as a Python user:

- ✓ comes with Intel Python framework

Advantages and disadvantages

DAAL in general:

- ✓ very simple installation/setup
- ✓ wide range of algorithms (both for machine l. and for deep l.)
- ✓ good support through dedicated intel forum (even for non paid versions)
- ✓ DAAL C++ can be called from R and Matlab (see how-to forum posts)
- ✗ documentation is sometimes not exhaustive
- ✗ examples cover very simple application cases

as a Python user:

- ✓ comes with Intel Python framework
- ✓ faster alternative to scikit

Advantages and disadvantages

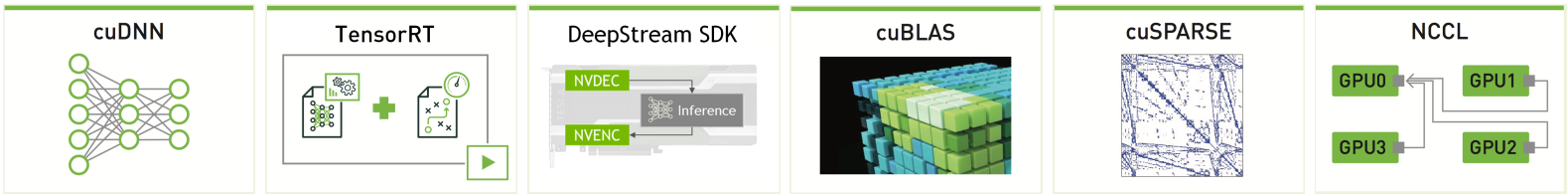
DAAL in general:

- ✓ very simple installation/setup
- ✓ wide range of algorithms (both for machine l. and for deep l.)
- ✓ good support through dedicated intel forum (even for non paid versions)
- ✓ DAAL C++ can be called from R and Matlab (see how-to forum posts)
- ✗ documentation is sometimes not exhaustive
- ✗ examples cover very simple application cases

as a Python user:

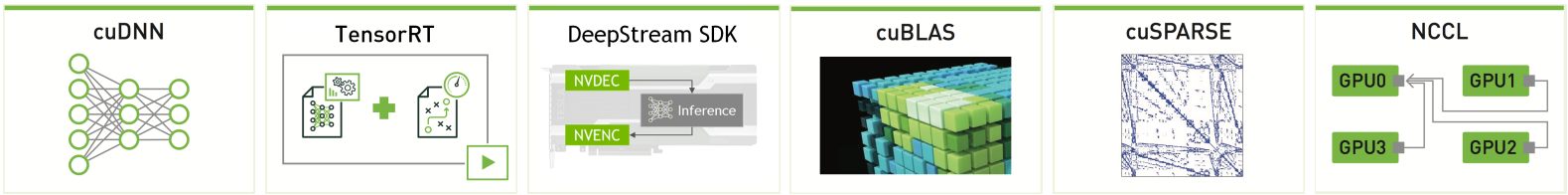
- ✓ comes with Intel Python framework
- ✓ faster alternative to scikit
- ✗ Python interface still in development phase
  - not all neural network layers parameters are accessible/modifiable

NVIDIA Deep Learning Software



Tools and libraries for designing and deploying GPU-accelerated deep learning applications.

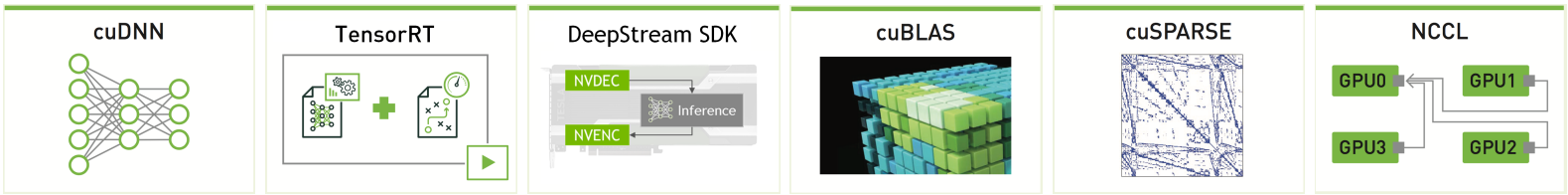
NVIDIA Deep Learning Software



Tools and libraries for designing and deploying GPU-accelerated deep learning applications.

- **Deep Learning Neural Network library (cuDNN)** forward and backward convolution, pooling, normalization, activation layers;

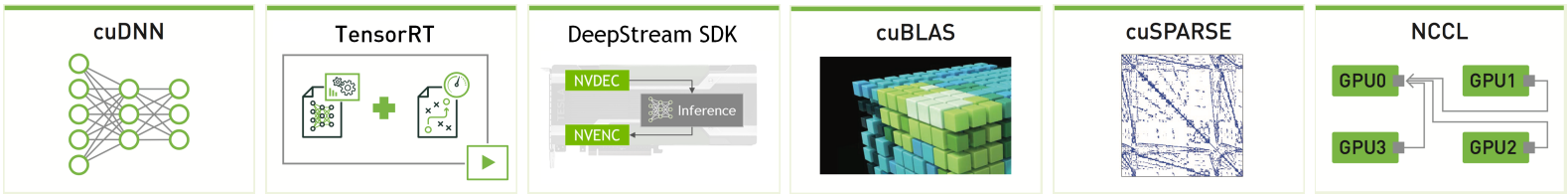
NVIDIA Deep Learning Software



Tools and libraries for designing and deploying GPU-accelerated deep learning applications.

- **Deep Learning Neural Network library (cuDNN)** forward and backward convolution, pooling, normalization, activation layers;
- **TensorRT** optimize, validate and deploy trained neural network for inference to hyperscale data centers, embedded, or automotive product platforms;

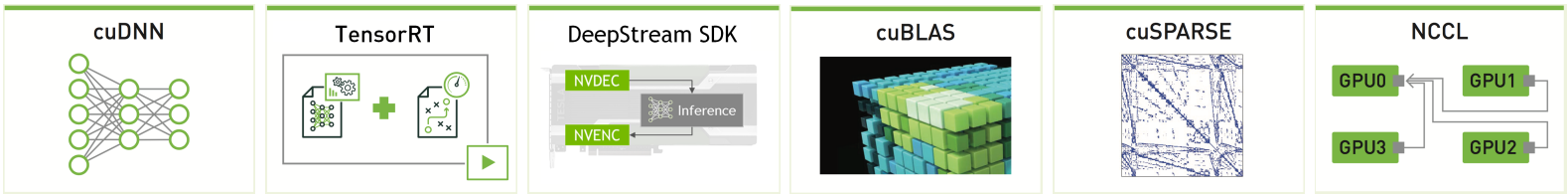
NVIDIA Deep Learning Software



Tools and libraries for designing and deploying GPU-accelerated deep learning applications.

- **Deep Learning Neural Network library (cuDNN)** forward and backward convolution, pooling, normalization, activation layers;
- **TensorRT** optimize, validate and deploy trained neural network for inference to hyperscale data centers, embedded, or automotive product platforms;
- **DeepStream SDK** uses TensorRT to deliver fast INT8 precision inference for real-time video content analysis, supports also FP16 and FP32 precisions;

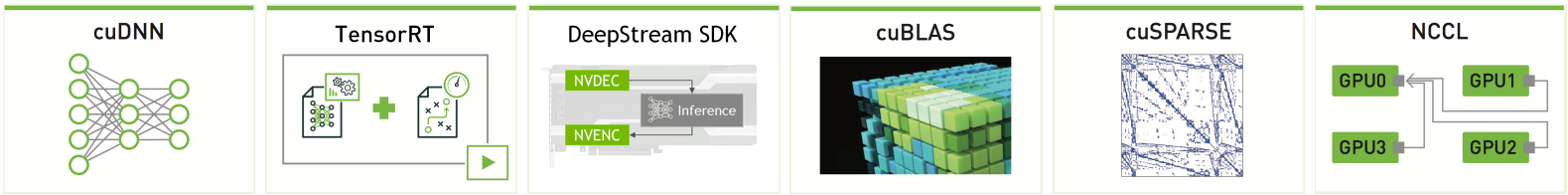
NVIDIA Deep Learning Software



Tools and libraries for designing and deploying GPU-accelerated deep learning applications.

- **Deep Learning Neural Network library (cuDNN)** forward and backward convolution, pooling, normalization, activation layers;
- **TensorRT** optimize, validate and deploy trained neural network for inference to hyperscale data centers, embedded, or automotive product platforms;
- **DeepStream SDK** uses TensorRT to deliver fast INT8 precision inference for real-time video content analysis, supports also FP16 and FP32 precisions;
- **Linear Algebra (cuBLAS and cuBLAS-XT)** accelerated BLAS subroutines for single and multi-GPU acceleration;

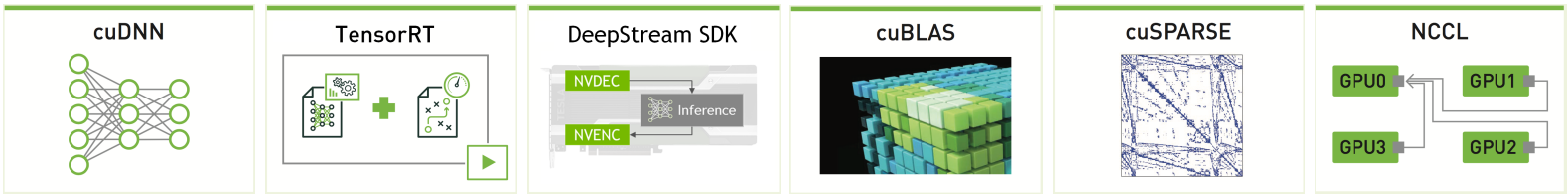
NVIDIA Deep Learning Software



Tools and libraries for designing and deploying GPU-accelerated deep learning applications.

- **Deep Learning Neural Network library (cuDNN)** forward and backward convolution, pooling, normalization, activation layers;
- **TensorRT** optimize, validate and deploy trained neural network for inference to hyperscale data centers, embedded, or automotive product platforms;
- **DeepStream SDK** uses TensorRT to deliver fast INT8 precision inference for real-time video content analysis, supports also FP16 and FP32 precisions;
- **Linear Algebra (cuBLAS and cuBLAS-XT)** accelerated BLAS subroutines for single and multi-GPU acceleration;
- **Sparse Linear Algebra (cuSPARSE)** supports dense, COO, CSR, CSC, ELL/HYB and Blocked CSR sparse matrix formats, Level 1,2,3 routines, sparse triangular solver, sparse tridiagonal solver;

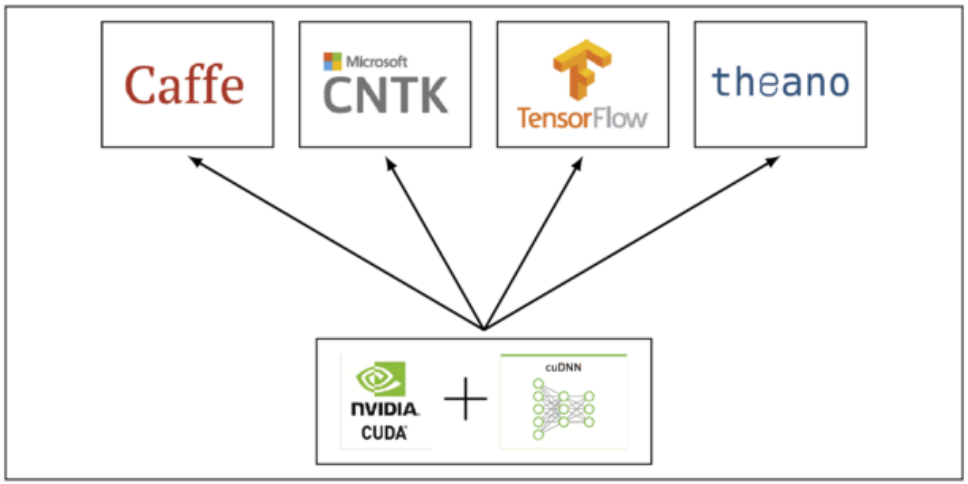
NVIDIA Deep Learning Software



Tools and libraries for designing and deploying GPU-accelerated deep learning applications.

- **Deep Learning Neural Network library (cuDNN)** forward and backward convolution, pooling, normalization, activation layers;
- **TensorRT** optimize, validate and deploy trained neural network for inference to hyperscale data centers, embedded, or automotive product platforms;
- **DeepStream SDK** uses TensorRT to deliver fast INT8 precision inference for real-time video content analysis, supports also FP16 and FP32 precisions;
- **Linear Algebra (cuBLAS and cuBLAS-XT)** accelerated BLAS subroutines for single and multi-GPU acceleration;
- **Sparse Linear Algebra (cuSPARSE)** supports dense, COO, CSR, CSC, ELL/HYB and Blocked CSR sparse matrix formats, Level 1,2,3 routines, sparse triangular solver, sparse tridiagonal solver;
- **Multi-GPU Communications (NCCL, pronounced ”Nickel”)** optimized primitives for collective multi-GPU communication;

NVIDIA based frameworks



TensorFlow

- Google Brain's second generation machine learning system

TensorFlow

- Google Brain's second generation machine learning system
- computations are expressed as stateful dataflow graphs

TensorFlow

- Google Brain's second generation machine learning system
- computations are expressed as stateful dataflow graphs
- automatic differentiation capabilities

TensorFlow

- Google Brain's second generation machine learning system
- computations are expressed as stateful dataflow graphs
- automatic differentiation capabilities
- optimization algorithms: gradient and proximal gradient based

TensorFlow

- Google Brain's second generation machine learning system
- computations are expressed as stateful dataflow graphs
- automatic differentiation capabilities
- optimization algorithms: gradient and proximal gradient based
- code portability (CPUs, GPUs, on desktop, server, or mobile computing platforms)

TensorFlow

- Google Brain's second generation machine learning system
- computations are expressed as stateful dataflow graphs
- automatic differentiation capabilities
- optimization algorithms: gradient and proximal gradient based
- code portability (CPUs, GPUs, on desktop, server, or mobile computing platforms)
- Python interface is the preferred one (Java and C++also exist)

TensorFlow

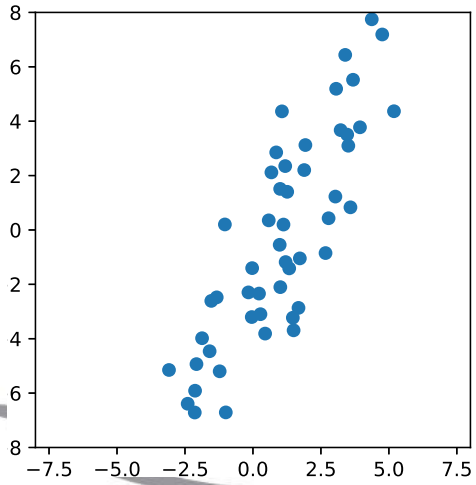
- Google Brain's second generation machine learning system
- computations are expressed as stateful dataflow graphs
- automatic differentiation capabilities
- optimization algorithms: gradient and proximal gradient based
- code portability (CPUs, GPUs, on desktop, server, or mobile computing platforms)
- Python interface is the preferred one (Java and C++also exist)
- installation through: virtualenv, pip, Docker, Anaconda, from sources

Linear regression (I)

```
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt

# Set up the data with a noisy linear relationship between X and Y.
num_examples = 50
X = np.array([np.linspace(-2, 4, num_examples), np.linspace(-6, 6,
     num_examples)])
X += np.random.randn(2, num_examples)
x, y = X
x_with_bias = np.array([(1., a) for a in x]).astype(np.float32)

losses = []
training_steps = 50
learning_rate = 0.002
```

Linear regression (I)

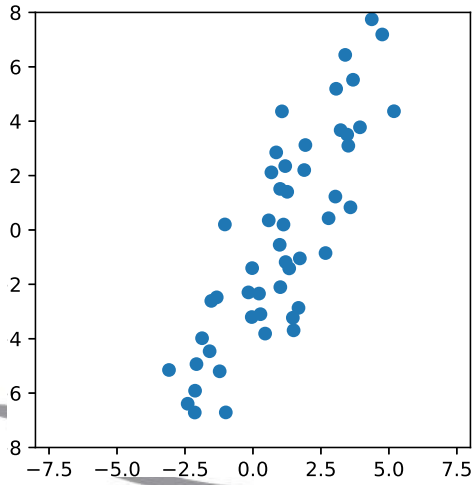
```
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt

# Set up the data with a noisy linear relationship between X and Y.
num_examples = 50
X = np.array([np.linspace(-2, 4, num_examples), np.linspace(-6, 6,
      num_examples)])
X += np.random.randn(2, num_examples)
x, y = X
x_with_bias = np.array([(1., a) for a in x]).astype(np.float32)

losses = []
training_steps = 50
learning_rate = 0.002
```



1. generate noisy input data

Linear regression (I)

```
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt

# Set up the data with a noisy linear relationship between X and Y.
num_examples = 50
X = np.array([np.linspace(-2, 4, num_examples), np.linspace(-6, 6,
      num_examples)])
X += np.random.randn(2, num_examples)
x, y = X
x_with_bias = np.array([(1., a) for a in x]).astype(np.float32)

losses = []
training_steps = 50
learning_rate = 0.002
```



1. generate noisy input data
2. set slack variables and fix algorithm parameters

Linear regression (II)

```
# Start of graph description
# Set up all the tensors, variables, and operations.
A = tf.constant(x_with_bias)
target = tf.constant(np.transpose([y]).astype(np.float32))
weights = tf.Variable(tf.random_normal([2, 1], 0, 0.1))

yhat = tf.matmul(A, weights)
yerror = tf.sub(yhat, target)
loss = tf.nn.l2_loss(yerror)

update_weights =
  tf.train.GradientDescentOptimizer(learning_rate).minimize(loss)

sess = tf.Session()
sess.run( tf.global_variables_initializer() )
for _ in range(training_steps):
  # Repeatedly run the operations, updating variables
  sess.run( update_weights )
  losses.append( sess.run( loss ) )
```

Linear regression (II)

```
# Start of graph description
# Set up all the tensors, variables, and operations.
A = tf.constant(x_with_bias)
target = tf.constant(np.transpose([y]).astype(np.float32))
weights = tf.Variable(tf.random_normal([2, 1], 0, 0.1))

yhat = tf.matmul(A, weights)
yerror = tf.sub(yhat, target)
loss = tf.nn.l2_loss(yerror)

update_weights =
  tf.train.GradientDescentOptimizer(learning_rate).minimize(loss)

sess = tf.Session()
sess.run( tf.global_variables_initializer() )
for _ in range(training_steps):
  # Repeatedly run the operations, updating variables
  sess.run( update_weights )
  losses.append( sess.run( loss ) )
```

3. define tensorflow constants and variables

Linear regression (II)

```
# Start of graph description
# Set up all the tensors, variables, and operations.
A = tf.constant(x_with_bias)
target = tf.constant(np.transpose([y]).astype(np.float32))
weights = tf.Variable(tf.random_normal([2, 1], 0, 0.1))

yhat = tf.matmul(A, weights)
yerror = tf.sub(yhat, target)
loss = tf.nn.l2_loss(yerror)

update_weights =
  tf.train.GradientDescentOptimizer(learning_rate).minimize(loss)

sess = tf.Session()
sess.run( tf.global_variables_initializer() )
for _ in range(training_steps):
  # Repeatedly run the operations, updating variables
  sess.run( update_weights )
  losses.append( sess.run( loss ) )
```

3. define tensorflow constants and variables
4. define nodes

Linear regression (II)

```
# Start of graph description
# Set up all the tensors, variables, and operations.
A = tf.constant(x_with_bias)
target = tf.constant(np.transpose([y]).astype(np.float32))
weights = tf.Variable(tf.random_normal([2, 1], 0, 0.1))

yhat = tf.matmul(A, weights)
yerror = tf.sub(yhat, target)
loss = tf.nn.l2_loss(yerror)

update_weights =
  tf.train.GradientDescentOptimizer(learning_rate).minimize(loss)

sess = tf.Session()
sess.run( tf.global_variables_initializer() )
for _ in range(training_steps):
  # Repeatedly run the operations, updating variables
  sess.run( update_weights )
  losses.append( sess.run( loss ) )
```

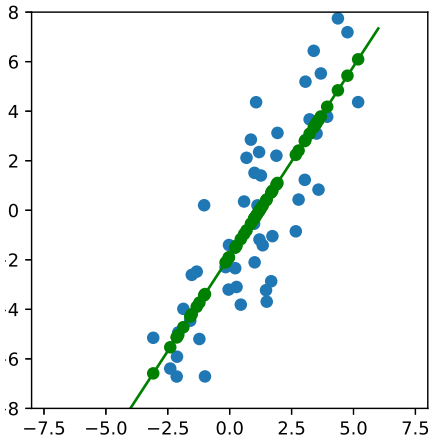
3. define tensorflow constants and variables
4. define nodes
5. start evaluation

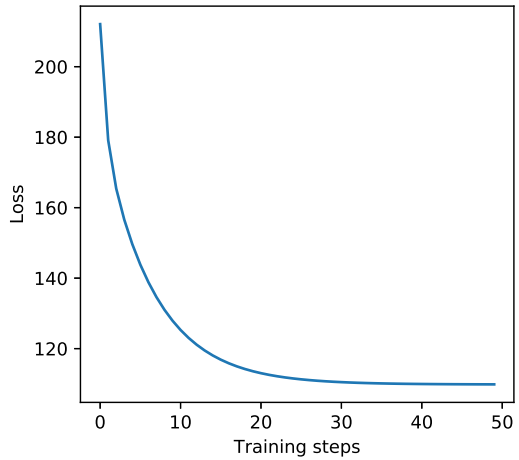
Linear regression (III)

```
# Training is done, get the final values
betas = sess.run( weights )
yhat = sess.run( yhat )
```

Linear regression (III)

```
# Training is done, get the final values
betas = sess.run( weights )
yhat = sess.run( yhat )
```



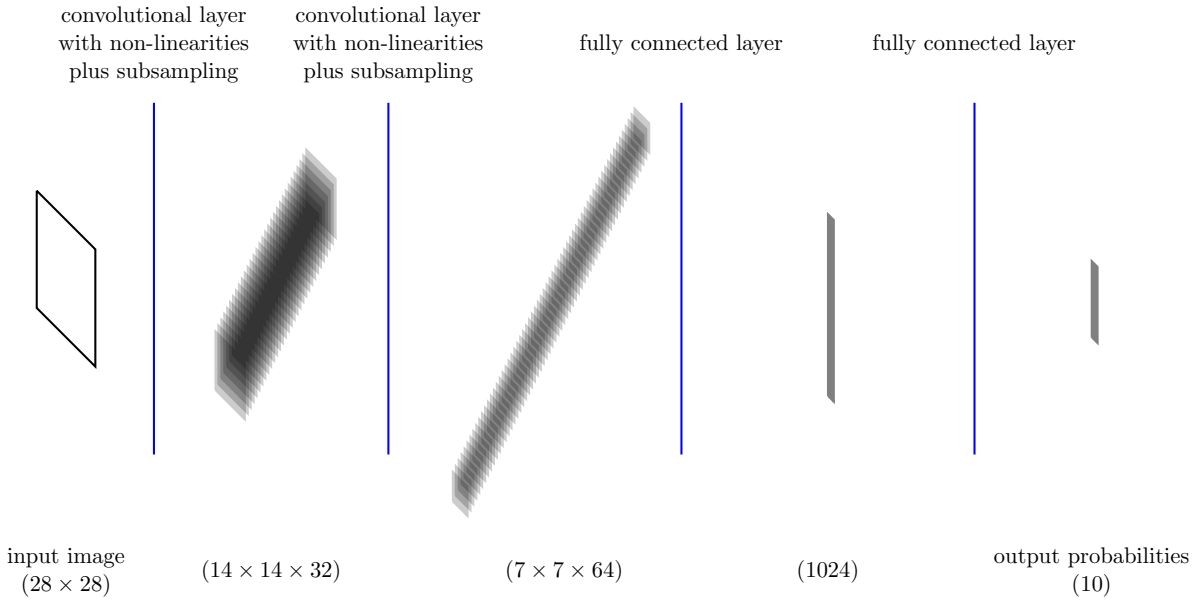


Benchmark results

- MNIST dataset of handwritten digits:
  - training set: 60k samples, 784 features per sample (MNIST)
  - test set: 10k samples, 784 features per sample (MNIST)

Benchmark results

- MNIST dataset of handwritten digits:
  - training set: 60k samples, 784 features per sample (MNIST)
  - test set: 10k samples, 784 features per sample (MNIST)
- Convolutional NN: two conv. layers, two fully conn. layers (plus reg.) $\approx$ 3m variables



Benchmark results (II)

Optimization method:

- stochastic gradient descent (batch size: 50 examples)
- fixed learning rate
- 2000 iterations

Benchmark results (II)

Optimization method:

- stochastic gradient descent (batch size: 50 examples)
- fixed learning rate
- 2000 iterations

Platforms:

- (1) Intel Core i5-6300U CPU @2.4GHz
- (2) 2 x Intel Xeon 2630 v3 @2.4GHz
- (3) Nvidia Tesla K40

Benchmark results (II)

Optimization method:

- stochastic gradient descent (batch size: 50 examples)
- fixed learning rate
- 2000 iterations

Platforms:

- (1) Intel Core i5-6300U CPU @2.4GHz
- (2) 2 x Intel Xeon 2630 v3 @2.4GHz
- (3) Nvidia Tesla K40

	PL (1)	PL (2)	PL (3)
training time [s]	3307.2	866.1	191.8
test time [s]	11.9	1.7	1.2

Benchmark results (II)

Optimization method:

- stochastic gradient descent (batch size: 50 examples)
- fixed learning rate
- 2000 iterations

Platforms:

- (1) Intel Core i5-6300U CPU @2.4GHz
- (2) 2 x Intel Xeon 2630 v3 @2.4GHz
- (3) Nvidia Tesla K40

	PL (1)	PL (2)	PL (3)
training time [s]	3307.2	866.1	191.8
test time [s]	11.9	1.7	1.2

Same code achieves 3.8x when running in 1 GALILEO node and 17.2x on a single GPU (training phase).

Advantages and disadvantages

TensorFlow in general:

Advantages and disadvantages

TensorFlow in general:

- ✓ very simple installation/setup

Advantages and disadvantages

TensorFlow in general:

- ✓ very simple installation/setup
- ✓ plenty of tutorials, exhaustive documentation

Advantages and disadvantages

TensorFlow in general:

- ✓ very simple installation/setup
- ✓ plenty of tutorials, exhaustive documentation
- ✓ tools for exporting (partially) trained graphs (see MetaGraph)

Advantages and disadvantages

TensorFlow in general:

- ✓ very simple installation/setup
- ✓ plenty of tutorials, exhaustive documentation
- ✓ tools for exporting (partially) trained graphs (see MetaGraph)
- ✓ debugger, graph flow visualization

Advantages and disadvantages

TensorFlow in general:

- ✓ very simple installation/setup
- ✓ plenty of tutorials, exhaustive documentation
- ✓ tools for exporting (partially) trained graphs (see MetaGraph)
- ✓ debugger, graph flow visualization

as a Python user:

- ✓ “Python API is the most complete and the easiest to use”

Advantages and disadvantages

TensorFlow in general:

- ✓ very simple installation/setup
- ✓ plenty of tutorials, exhaustive documentation
- ✓ tools for exporting (partially) trained graphs (see MetaGraph)
- ✓ debugger, graph flow visualization

as a Python user:

- ✓ “Python API is the most complete and the easiest to use”
- ✓ numpy interoperability

Advantages and disadvantages

TensorFlow in general:

- ✓ very simple installation/setup
- ✓ plenty of tutorials, exhaustive documentation
- ✓ tools for exporting (partially) trained graphs (see MetaGraph)
- ✓ debugger, graph flow visualization

as a Python user:

- ✓ "Python API is the most complete and the easiest to use"
- ✓ numpy interoperability
- ✗ lower level than pyDAAL (?)