Deep Learning Frameworks
Caffe and Caffe 2

Presenter :: Shivani Sabhlok
GIRL, I KNOW

YOU DON'T EVEN HAVE TO SAY ANYTHING
I AM SERIOUS

JUST KIDDING

Source: http://catplanet.org/i-am-serious-just-kidding-cat-meme/
The **ImageNet** project is a large visual database designed for use in visual object recognition software research.
The ImageNet project runs an annual software contest, the ImageNet Large Scale Visual Recognition Challenge (ILSVRC), where software programs compete to correctly classify and detect objects and scenes.
Recap

CNN architectures of ILSVRC top competitors

LeNet-5

- pioneered 7-level convolutional network to recognise hand-written numbers on checks (cheques)
- 32x32 pixel greyscale input images
- This technique is constrained by the availability of computing resources.

• significantly outperformed all the prior competitors
• won the challenge in 2012 by reducing the top-5 error from 26% to 15.3%
• The network had a very similar architecture as LeNet but was deeper, with more filters per layer, and with stacked convolutional layers.
• It consisted 11x11, 5x5,3x3, convolutions, max pooling, dropout, data augmentation, ReLU activations, SGD with momentum
• was trained for 6 days simultaneously on two Nvidia GeForce GTX 580 GPUs

Source: https://medium.com/@sidereal/cnns-architectures-leenet-alexnet-vgg-googlenet-resnet-and-more-666091488d5
ZFNet

- ILSVRC 2013 winner
- It achieved a top-5 error rate of 14.8%
- It was mostly an achievement by tweaking the hyper-parameters of AlexNet while maintaining the same structure with additional Deep Learning elements

• winner of the ILSVRC 2014 competition
• achieved a top-5 error rate of 6.67%
• This was very close to human level performance which the organisers of the challenge were now forced to evaluate
• implemented a novel element which is dubbed an inception module
• consisted of a 22 layer deep CNN but reduced the number of parameters from 60 million (AlexNet) to 4 million
ResNet

• introduced a novel architecture with “skip connections” and features heavy batch normalization.
• skip connections are also known as gated units or gated recurrent units and have a strong similarity to recent successful elements applied in RNN
• It achieves a top-5 error rate of 3.57% which beats human-level performance on this dataset.

Do we need a Deep Learning Framework??

Do we need a Deep Learning Framework??

Well.. I sort of understand what DL is but what is a framework?!?!?!

Do we need a Deep Learning Framework??

Source: https://www.youtube.com/watch?v=jHHT1js-UOQ
Do we need a Deep Learning Framework??

Expectations from a DL framework

Source: https://www.memecenter.com/fun/4848063/nsfw/comments
Expectations from a DL framework

(1) Easily build big computational graphs (2) Easily compute gradients in computational graphs (3) Run it all efficiently on GPU (wrap cuDNN, cuBLAS, etc)

(2) These frameworks allow separation of representation from actual implementation and allow seamless switching between CPU and GPU
How does a DL Framework help?

By separating model representation from actual implementation, DL frameworks allows experimentation and seamless switching among platforms for ease of development and deployment from prototyping machines to cloud environments.

Caffe – modifiable framework and a collection of reference models for deep learning

Source ::
What Caffe does

• Caffe provides a complete toolkit for training, testing, finetuning, and deploying models, with well-documented examples for all of these tasks.

• Alexnet and imagenet cnn are available out of the box
How it does that

- Modular design :: easy extension to data formats, network layers, loss functions with lot of these available out of the box

- Allows binding with python and matlab for to construct and classify

- Provides clear access to deep architectures. The code is written in clean, efficient C++, with CUDA used for GPU computation, and nearly complete, well-supported bindings to Python/Numpy and MATLAB.
DeCaf Options ??

<table>
<thead>
<tr>
<th>Framework</th>
<th>License</th>
<th>Core language</th>
<th>Binding(s)</th>
<th>CPU</th>
<th>GPU</th>
<th>Open source</th>
<th>Training</th>
<th>Pretrained models</th>
<th>Development</th>
</tr>
</thead>
<tbody>
<tr>
<td>Caffe</td>
<td>BSD</td>
<td>C++</td>
<td>Python, MATLAB</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>distributed</td>
</tr>
<tr>
<td>cuda-convnet [7]</td>
<td>unspecified</td>
<td>C++</td>
<td>Python</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>discontinued</td>
</tr>
<tr>
<td>Decaf [2]</td>
<td>BSD</td>
<td>Python</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>discontinued</td>
</tr>
<tr>
<td>OverFeat [9]</td>
<td>unspecified</td>
<td>Lua</td>
<td>C++,Python</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>centralized</td>
</tr>
<tr>
<td>Theano/Pylearn2 [4]</td>
<td>BSD</td>
<td>Python</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>distributed</td>
</tr>
<tr>
<td>Torch7 [1]</td>
<td>BSD</td>
<td>Lua</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>distributed</td>
</tr>
</tbody>
</table>

Table 1: Comparison of popular deep learning frameworks. Core language is the main library language, while bindings have an officially supported library interface for feature extraction, training, etc. CPU indicates availability of host-only computation, no GPU usage (e.g., for cluster deployment); GPU indicates the GPU computation capability essential for training modern CNNs.

Source ::
Upcoming keywords overview

Google Protocol Buffers

```protobuf
message Person {
  required string name = 1;
  required int32 id = 2;
  optional string email = 3;
}
```
Upcoming keywords overview

Google Protocol Buffers

LeveDB
Key terms and concepts

• Models are saved to disc as google protocol buffers

• Large scale data is stored in the levelDB

• Data is stored and communicated in 4 dimensional arrays called blobs
About Caffe: Key terms and concepts

- A Caffe layer: the essence of a neural network layer
- Model definitions: config files using protocol buffer language
Good things

Modular
Good things

Modular

Separation of representation and implementation
Good things

- Modular
- Separation of representation and implementation
- Python and MATLAB bindings
Good things

- Modular
- Separation of representation and implementation
- Python and MATLAB bindings
- Pre-trained reference models

Source: https://sayingimages.com/thumbs-up-meme/
Caffe major pieces

• **Nets, Layers, and Blobs**: the anatomy of a Caffe model.
• **Forward / Backward**: the essential computations of layered compositional models.
• **Loss**: the task to be learned is defined by the loss.
• **Solver**: the solver coordinates model optimization.
• **Layer Catalogue**: fundamental unit of modeling and computation – Caffe’s catalogue includes layers for state-of-the-art models.
• **Interfaces**: command line, Python, and MATLAB Caffe.
• **Data**: how to caffeinate data for model input.

Source: [http://caffe.berkeleyvision.org/tutorial/](http://caffe.berkeleyvision.org/tutorial/)
• Nets, Layers, and Blobs: the anatomy of a Caffe model.
Each layer type defines three critical computations: setup, forward, and backward.

Setup: initialize the layer and its connections once at model initialization.
Forward: given input from bottom compute the output and send to the top.
Backward: given the gradient w.r.t. the top output compute the gradient w.r.t. to the input and send to the bottom. A layer with parameters computes the gradient w.r.t. to its parameters and stores it internally.

Source: http://caffe.berkeleyvision.org/tutorial/
A simple logistic regression classifier is defined by the protobuf schema in caffe.proto. The source file is mostly self-explanatory.

Source: [http://caffe.berkeleyvision.org/tutorial/](http://caffe.berkeleyvision.org/tutorial/)
Caffe major pieces

- **Nets, Layers, and Blobs**: the anatomy of a Caffe model.
- **Forward / Backward**: the essential computations of layered compositional models.
- **Loss**: the task to be learned is defined by the loss.
- **Solver**: the solver coordinates model optimization.
- **Layer Catalogue**: fundamental unit of modeling and computation – Caffe’s catalogue includes layers for state-of-the-art models.
- **Interfaces**: command line, Python, and MATLAB Caffe.
- **Data**: how to caffeinate data for model input.

Source: [http://caffe.berkeleyvision.org/tutorial/](http://caffe.berkeleyvision.org/tutorial/)
Forward / Backward: the essential computations of layered compositional models.

Source: [http://caffe.berkeleyvision.org/tutorial/](http://caffe.berkeleyvision.org/tutorial/)
Let’s consider a simple logistic regression classifier. The **forward** pass computes the output given the input for inference. In forward Caffe composes the computation of each layer to compute the “function” represented by the model. This pass goes from bottom to top.

Source: [http://caffe.berkeleyvision.org/tutorial/](http://caffe.berkeleyvision.org/tutorial/)
The **backward** pass computes the gradient given the loss for learning. In backward Caffe reverse-composes the gradient of each layer to compute the gradient of the whole model by automatic differentiation. This is back-propagation. This pass goes from top to bottom.

Source: [http://caffe.berkeleyvision.org/tutorial/](http://caffe.berkeleyvision.org/tutorial/)
Caffe major pieces

• **Nets, Layers, and Blobs**: the anatomy of a Caffe model.
• **Forward / Backward**: the essential computations of layered compositional models.
• **Loss**: the task to be learned is defined by the loss.
• **Solver**: the solver coordinates model optimization.
• **Layer Catalogue**: fundamental unit of modeling and computation – Caffe’s catalogue includes layers for state-of-the-art models.
• **Interfaces**: command line, Python, and MATLAB Caffe.
• **Data**: how to caffeinate data for model input.

Source: [http://caffe.berkeleyvision.org/tutorial/](http://caffe.berkeleyvision.org/tutorial/)
The loss in Caffe is computed by the Forward pass of the network. Each layer takes a set of input (bottom) blobs and produces a set of output (top) blobs. Some of these layers’ outputs may be used in the loss function. A typical choice of loss function for one-versus-all classification tasks is the SoftmaxWithLoss function, used in a network definition as follows, for example:

```
layer {
  name: "loss"
  type: "SoftmaxWithLoss"
  bottom: "pred"
  bottom: "label"
  top: "loss"
  loss_weight: 1
}
```

In a SoftmaxWithLoss function, the top blob is a scalar (empty shape) which averages the loss (computed from predicted labels pred and actuals labels label) over the entire mini-batch.

Source :: http://caffe.berkeleyvision.org/tutorial/
Caffe major pieces

- **Nets, Layers, and Blobs**: the anatomy of a Caffe model.
- **Forward / Backward**: the essential computations of layered compositional models.
- **Loss**: the task to be learned is defined by the loss.
- **Solver**: the solver coordinates model optimization.
- **Layer Catalogue**: fundamental unit of modeling and computation – Caffe’s catalogue includes layers for state-of-the-art models.
- **Interfaces**: command line, Python, and MATLAB Caffe.
- **Data**: how to caffeinate data for model input.

Source: [http://caffe.berkeleyvision.org/tutorial/](http://caffe.berkeleyvision.org/tutorial/)
Solver: the solver coordinates model optimization

The Caffe solvers are:
• Stochastic Gradient Descent (type: "SGD"),
• AdaDelta (type: "AdaDelta"),
• Adaptive Gradient (type: "AdaGrad"),
• Adam (type: "Adam"),
• Nesterov’s Accelerated Gradient (type: "Nesterov") and
• RMSprop (type: "RMSProp")

Source : http://caffe.berkeleyvision.org/tutorial/
Caffe major pieces

• **Nets, Layers, and Blobs**: the anatomy of a Caffe model.

• **Forward / Backward**: the essential computations of layered compositional models.

• **Loss**: the task to be learned is defined by the loss.

• **Solver**: the solver coordinates model optimization.

• **Layer Catalogue**: fundamental unit of modeling and computation – Caffe’s catalogue includes layers for state-of-the-art models.

• **Interfaces**: command line, Python, and MATLAB Caffe.

• **Data**: how to caffeinate data for model input.

Source: [http://caffe.berkeleyvision.org/tutorial/](http://caffe.berkeleyvision.org/tutorial/)
Layer Catalogue: the layer is the fundamental unit of modeling and computation – Caffe’s catalogue includes layers for state-of-the-art models.

To create a Caffe model you need to define the model architecture in a protocol buffer definition file (prototxt). Caffe layers and their parameters are defined in the protocol buffer definitions for the project in caffe.proto.

Kinds of layers:
- Data layers
- Vision layers
- Recurrent layers
- Common layers
- Normalization layers
- Activation/ neuron layers
- Utility layers
- Loss layers

Source: http://caffe.berkeleyvision.org/tutorial/
Data layers

Layers:

- **Image Data** - read raw images.
- **Database** - read data from LEVELDB or LMDB.
- **HDF5 Input** - read HDF5 data, allows data of arbitrary dimensions.
- **HDF5 Output** - write data as HDF5.
- **Input** - typically used for networks that are being deployed.
- **Window Data** - read window data file.
- **Memory Data** - read data directly from memory.
- **Dummy Data** - for static data and debugging.

Source: [http://caffe.berkeleyvision.org/tutorial/](http://caffe.berkeleyvision.org/tutorial/)
Vision Layers

Layers:

- **Convolution Layer** - convolves the input image with a set of learnable filters, each producing one feature map in the output image.
- **Pooling Layer** - max, average, or stochastic pooling.
- **Spatial Pyramid Pooling (SPP)**
- **Crop** - perform cropping transformation.
- **Deconvolution Layer** - transposed convolution.

Source: [http://caffe.berkeleyvision.org/tutorial/](http://caffe.berkeleyvision.org/tutorial/)
Utility Layers

Layers:

- Flatten
- Reshape
- Batch Reindex
- Split
- Concat
- Slicing
- Eltwise - element-wise operations such as product or sum between two blobs.
- Filter / Mask - mask or select output using last blob.
- Parameter - enable parameters to be shared between layers.
- Reduction - reduce input blob to scalar blob using operations such as sum or mean.
- Silence - prevent top-level blobs from being printed during training.
- ArgMax
- Softmax
- Python - allows custom Python layers.

Common Layers

Layers:

- Inner Product - fully connected layer.
- Dropout
- Embed - for learning embeddings of one-hot encoded vector (takes index as input).

Recurrent Layers

Layers:

- Recurrent
- RNN
- Long-Short Term Memory (LSTM)

Normalization Layers

Layers:

- Local Response Normalization (LRN) - performs a kind of "lateral inhibition" by normalizing over local input regions.
- Mean Variance Normalization (MVN) - performs contrast normalization / instance normalization.
- Batch Normalization - performs normalization over mini-batches.

The bias and scale layers can be helpful in combination with normalization.

Source: http://caffe.berkeleyvision.org/tutorial/
Activation/ neuron layers

 Layers:

- ReLU / Rectified-Linear and Leaky-ReLU - ReLU and Leaky-ReLU rectification.
- PReLU - parametric ReLU.
- ELU - exponential linear rectification.
- Sigmoid
- TanH
- Absolute Value
- Power - \( f(x) = (\text{shift} + \text{scale} \times x)^{\text{power}} \).
- Exp - \( f(x) = \text{base}^{(\text{shift} + \text{scale} \times x)} \).
- Log - \( f(x) = \log(x) \).
- BNLL - \( f(x) = \log(1 + \exp(x)) \).
- Threshold - performs step function at user defined threshold.
- Bias - adds a bias to a blob that can either be learned or fixed.
- Scale - scales a blob by an amount that can either be learned or fixed.

Source: [http://caffe.berkeleyvision.org/tutorial/](http://caffe.berkeleyvision.org/tutorial/)

Loss layers

Layers:

- **Multinomial Logistic Loss**
- **InfoGain Loss** - a generalization of MultinomialLogisticLossLayer.
- **Softmax with Loss** - computes the multinomial logistic loss of the softmax of its inputs. It's conceptually identical to a softmax layer followed by a multinomial logistic loss layer, but provides a more numerically stable gradient.
- **Sum-of-Squares / Euclidean** - computes the sum of squares of differences of its two inputs.
  \[
  \frac{1}{2N} \sum_{i=1}^{N} \| x_i^1 - x_i^2 \|^2_2
  \]
- **Hinge / Margin** - The hinge loss layer computes a one-vs-all hinge (L1) or squared hinge loss (L2).
- **Sigmoid Cross-Entropy Loss** - computes the cross-entropy (logistic) loss, often used for predicting targets interpreted as probabilities.
- **Accuracy / Top-k layer** - scores the output as an accuracy with respect to target – it is not actually a loss and has no backward step.
- **Contrastive Loss**

Source: [http://caffe.berkeleyvision.org/tutorial/](http://caffe.berkeleyvision.org/tutorial/)
Caffe major pieces

- **Nets, Layers, and Blobs**: the anatomy of a Caffe model.
- **Forward / Backward**: the essential computations of layered compositional models.
- **Loss**: the task to be learned is defined by the loss.
- **Solver**: the solver coordinates model optimization.
- **Layer Catalogue**: fundamental unit of modeling and computation – Caffe’s catalogue includes layers for state-of-the-art models.
- **Interfaces**: command line, Python, and MATLAB Caffe.
- **Data**: how to caffeinate data for model input.

Source: [http://caffe.berkeleyvision.org/tutorial/](http://caffe.berkeleyvision.org/tutorial/)
This data layer definition

```plaintext
layer {
  name: "mnist"
  # Data layer loads levelDB or lmdb storage DBs for high-throughput.
  type: "Data"
  # the 1st top is the data itself; the name is only convention
  top: "data"
  # the 2nd top is the ground truth; the name is only convention
  top: "label"
  # the data layer configuration
  data_param {
    # path to the DB
    source: "examples/mnist/mnist_train_lmdb"
    # type of DB: LEVELDB or LMDB (LMDB supports concurrent reads)
    backend: LMDB
    # batch processing improves efficiency.
    batch_size: 64
  }
  # common data transformations
  transform_param {
    # feature scaling coefficient: this maps the [0, 255] MNIST data to [0, 1]
    scale: 0.00390625
    } 
}
```
loads the MNIST digits.

Source: [http://caffe.berkeleyvision.org/tutorial/](http://caffe.berkeleyvision.org/tutorial/)
SHOW ME THE CODE

AND I NEED IT NOW

YEAH, IF YOU COULD JUST SHOW ME SOME CODE

THAT'D BE GREAT
1. Define the model (lenet_train_test.prototxt)

scale the incoming pixels so that they are in the range 
\([0,1)\)
0.00390625? It is 1 divided by 256.
1. Define the model (lenet_train_test.prototxt)

scale the incoming pixels so that they are in the range [0,1) 0.00390625? It is 1 divided by 256.

set the weight learning rate to be the same as the learning rate given by the solver during runtime, and the bias learning rate to be twice as large as that.

xavier algorithm automatically determines the scale of initialization based on the number of input and output neurons the default filling value 0.

Source: https://github.com/BVLC/caffe/blob/master/examples/mnist/lenet_train_test.prototxt
http://caffe.berkeleyvision.org/gathered/examples/mnist.html
1. Define the model (lenet_train_test.prototxt)

- Scale the incoming pixels so that they are in the range [0,1) 0.00390625? It is 1 divided by 256.

- Set the weight learning rate to be the same as the learning rate given by the solver during runtime, and the bias learning rate to be twice as large as that.

- Xavier algorithm automatically determines the scale of initialization based on the number of input and output neurons.

- The default filling value 0.

Similarly, you can write the Fully connected layer, ReLU layer, Loss layer etc.
1. Define the model (lenet_train_test.prototxt)

scale the incoming pixels so that they are in the range [0,1) 0.00390625? It is 1 divided by 256.

set the weight learning rate to be the same as the learning rate given by the solver during runtime, and the bias learning rate to be twice as large as that.

xavier algorithm automatically determines the scale of initialization based on the number of input and output neurons the default filling value 0.

Similarly you can write the Fully connected layer, ReLU layer, Loss layer etc.

Layer definitions can include rules for whether and when they are included in the network definition.

Source: https://github.com/BVLC/caffe/blob/master/examples/mnist/lenet_train_test.prototxt
http://caffe.berkeleyvision.org/gathered/examples/mnist.html
2. Define the solver (lenet_solver.prototxt)

```
# The train/test net protocol buffer definition
net: "examples/mnist/lenet_train_test.prototxt"
# test_iter specifies how many forward passes the test should carry out.
# In the case of MNIST, we have test batch size 100 and 100 test iterations,
# covering the full 10,000 testing images.
test_iter: 100
# Carry out testing every 500 training iterations.
test_interval: 500
# The base learning rate, momentum and the weight decay of the network.
base_lr: 0.01
momentum: 0.9
weight_decay: 0.0005
# The learning rate policy
lr_policy: "inv"
gamma: 0.0001
power: 0.75
# Display every 100 iterations
display: 100
# The maximum number of iterations
max_iter: 10000
# Display intermediate results
snapshot: 5000
snapshot_prefix: "examples/mnist/lenet"
# solver mode: CPU or GPU
solver_mode: GPU
```
Hey girl

I know it's hard, but you can do it. I believe in you.
from caffe import layers as L, params as P

def lenet(ldb, batch_size):
    # our version of LeNet: a series of linear and simple nonlinear transformations
    n = caffe.NetSpec()

    n.data, n.label = L.Data(batch_size=batch_size, backend=P.Data.LMDB, source=ldb,
                             transform_param=P.Data.transform_param(scaling=1./255), ntop=2)

    n.conv1 = L.Convolution(n.data, kernel_size=5, num_output=20, weight_filler=dict(type='xavier'))
    n.pool1 = L.Pooling(n.conv1, kernel_size=2, stride=2, pool=P.Pooling.MAX)

    n.conv2 = L.Convolution(n.pool1, kernel_size=5, num_output=50, weight_filler=dict(type='xavier'))
    n.pool2 = L.Pooling(n.conv2, kernel_size=2, stride=2, pool=P.Pooling.MAX)

    n.fc1 = L.InnerProduct(n.pool2, num_output=500, weight_filler=dict(type='xavier'))
    n.relu1 = L.ReLU(n.fc1, in_place=True)
    n.score = L.InnerProduct(n.relu1, num_output=10, weight_filler=dict(type='xavier'))
    n.loss = L.SoftmaxWithLoss(n.score, n.label)

    return n.to_proto()

with open('mnist/lenet_auto_train.prototxt', 'w') as f:
    f.write(str(lenet('mnist/mnist_train_lmdb', 64)))

with open('mnist/lenet_auto_test.prototxt', 'w') as f:
    f.write(str(lenet('mnist/mnist_test_lmdb', 100)))
Source: https://www.eastbaytimes.com/2017/07/31/electric-train-stolen-from-pleasant-hill-found-intact/
Caffe summary

- Applications in machine learning, vision, speech and multimedia
- Good for feed-forward networks and image processing
- Excellent ConvNet implementation

- Not intended for applications such as text, sound or time series data.

Caffe- Interface

- mainly: command line interface
- supports also pycaffe interface
- model: defined in protobuf - using a text editor
- even if you use pycaffe

Caffe- Model examples

layer {
  name: "pool1"
  type: "Pooling"
  pooling_param {
    kernel_size: 2
    stride: 2
    pool: MAX
  }
  bottom: "data"
  top: "pool1"
}

layer {
  name: "conv1"
  type: "Convolution"
  bottom: "data"
  top: "conv1"
  param {
    lr_mult: 0
    decay_mult: 0
  }
}

classification_param {
  num_output: 64
  kernel_size: 3
  pad: 1
}

layer {
  name: "loss"
  type: "SoftmaxWithLoss"
  bottom: "fc8"
  bottom: "label"
  top: "loss"
}

Caffe- Solver

- creation of the training network for learning and test network(s) for evaluation
- iterative optimization by calling forward / backward and parameter updating
- (periodical) evaluation of the test networks
- snapshotting of the model and solver state throughout the optimization

Caffe- Solver example

base_lr: 0.01
# begin training at a learning rate of 0.01 - 1e-2
lr_policy: "step"
# learning rate policy: drop the learning rate in "steps" by a factor of gamma every stepsize iterations
gamma: 0.1
# drop the learning rate by a factor of 0.1
# (i.e., multiply it by a factor of gamma = 0.1)
stepsize: 100000
# drop the learning rate every 100K iterations
max_iter: 350000
# train for 350K iterations total
Caffe- Architecture

When born: excellent, nowadays: average

- Need to write C++/CUDA code for new GPU layers
- Need to define the full forward, backward and gradient update for new layers
- Need to implement extra functions for both CPU/GPU support (eg. Forward_gpu, Forward_cpu)

Caffe- Extra

- Not good for RNNs, mainly CNNs
- Cumbersome for big networks (eg., GoogLeNet, ResNet)
WHAT DO YOU BRING TO THE TABLE?
Caffe2 improves Caffe 1.0 in a series of directions:

- first-class support for large-scale distributed training
- mobile deployment
- new hardware support (in addition to CPU and CUDA) (server CPU, GPU, iOS, and Android)
- flexibility for future directions such as quantized computation
- stress tested by the vast scale of Facebook applications
The algorithm developer can focus on the algorithm work and not on how to run the convolution.
One of basic units of computation in Caffe2 are the Operators.

Each operator contains the logic necessary to compute the output given the appropriate number and type of inputs and parameters. The overall difference between operators’ functionality in Caffe and Caffe2 is illustrated in the following graphic, respectively:

IT'S SHOWTIME FOLKS
```python
X = np.random.randn(2, 3)
W = np.random.randn(5, 3)
b = np.ones(5)
Y = X * W^T + b
```
```python
X = np.random.randn(2, 3)
W = np.random.randn(5, 3)
b = np.ones(5)
Y = X * W.T + b
```

```python
net = core.Net("my_first_net")
```
Let's create a blob called X, and use GaussianFill to fill it with some random data.

```
x = net.GaussianFill([], ["x"], mean=0.0, std=1.0, shape=[2, 3], run_once=0)
```
Let's create a blob called X, and use GaussianFill to fill it with some random data.

```python
X = net.GaussianFill([], ['X'], mean=0.0, std=1.0, shape=[2, 3], run_once=0)
```

```python
W = net.GaussianFill([], ['W'], mean=0.0, std=1.0, shape=[5, 3], run_once=0)
b = net.ConstantFill([], ['b'], shape=[5,], value=1.0, run_once=0)
```
since the BlobReference objects know what net it is generated from, in addition to creating operators from net, you can also create operators from BlobReferences
FC :: Fully connected layer
The layer computes $Y = X \cdot W^T + b$, where $X$ has size $(M \times K)$, $W$ has size $(N \times K)$, $b$ has size $(N)$, and $Y$ has size $(M \times N)$, where $M$ is often the batch size.
\[ Y = \text{FC}([W, b], ["y"]) \]

```
name: "my_first_net"

op {
  output: "X"
  name: ""n
  type: "GaussianFill"
  arg {
    name: "std"
    f: 1.0
  }
  arg {
    name: "run_once"
    i: 0
  }
  arg {
    name: "shape"
    ints: 2
    ints: 3
  }
  arg {
    name: "mean"
    f: 0.0
  }
}

op {
  output: "W"
  name: ""
  type: "GaussianFill"
  arg {
    name: "std"
    f: 1.0
  }
  arg {
    name: "run_once"
    i: 0
  }
  arg {
    name: "shape"
    ints: 5
    ints: 3
  }
  arg {
    name: "mean"
    f: 0.0
  }
}

op {
  output: "b"
  name: ""
  type: "ConstantFill"
  arg {
    name: "run_once"
    i: 0
  }
  arg {
    name: "shape"
    ints: 5
  }
  arg {
    name: "value"
    f: 1.0
  }
}
```

```
Y = \text{FC}([X, W, b], ["y"]) 
```