Deep Transfer Learning on Caffe

Eric Fan

2017/11/22

yfanal@connect.ust.hk

Outline

- Basics of Caffe
- Deep Transfer Learning on Caffe

Caffe

Caffe is a deep learning framework made with expression, speed, and modularity in mind. It is developed by Berkeley AI Research (BAIR) and by community contributors. Yangqing Jia created the project during his PhD at UC Berkeley. Caffe is released under the BSD 2-Clause license.

Caffe Featues

- Pure C++ / CUDA library for deep learning Command line, Python, MATLAB interfaces
- Fast, well-tested code
- Tools, reference models, demos, and recipes
- Seamless switch between CPU and GPU
- In general, Caffe allow you to DIY your own DL application based on configuration rather than code.

Install caffe

Install Dependancies(RHEL/Fedora/CentOS)

- General Dependancies:

  ```
  sudo yum install protobuf-devel leveldb-devel snappy-devel opencv-devel boost-devel hdf5-devel
  ```

- Remaining dependencies:

  ```
  sudo yum install gflags-devel glog-devel lmdb-devel
  ```

- Remaining dependencies, if not found `glog`:

  ```
  wget https://storage.googleapis.com/google-code-archive-downloads/v2/code.google.com/google-glog/glog-0.3.3.tar.gz
  tar zxvf glog-0.3.3.tar.gz
  cd glog-0.3.3
  ./configure
  make && make install
  ```

Installation

Install Dependancies(RHEL/Fedora/CentOS)

- Remaining dependencies, if not found `gflags`:

  `wget`

  `https://github.com/schuhschuh/gflags/archive/master.zip`

  `unzip master.zip cd gflags-master mkdir build && cd build`

  `export CXXFLAGS="-fPIC" && cmake .. && make VERBOSE=1 make`

  `&& make install`

- Remaining dependencies, if not found `lmdb`:

  `git clone https://github.com/LMDB/lmdb cd`

  `lmdb/libraries/liblmdb`

  `make && make install`

Installation

Compilation

Caffe can be compiled with either Make or CMake. Make is officially supported while CMake is supported by the community.

- Configure makefile (For example, if using Anaconda Python, or if cuDNN is desired)

  `cp Makefile.config.example Makefile.config`

- make

  `make all`

  `make test`

  `make runtest`

Makefile.config

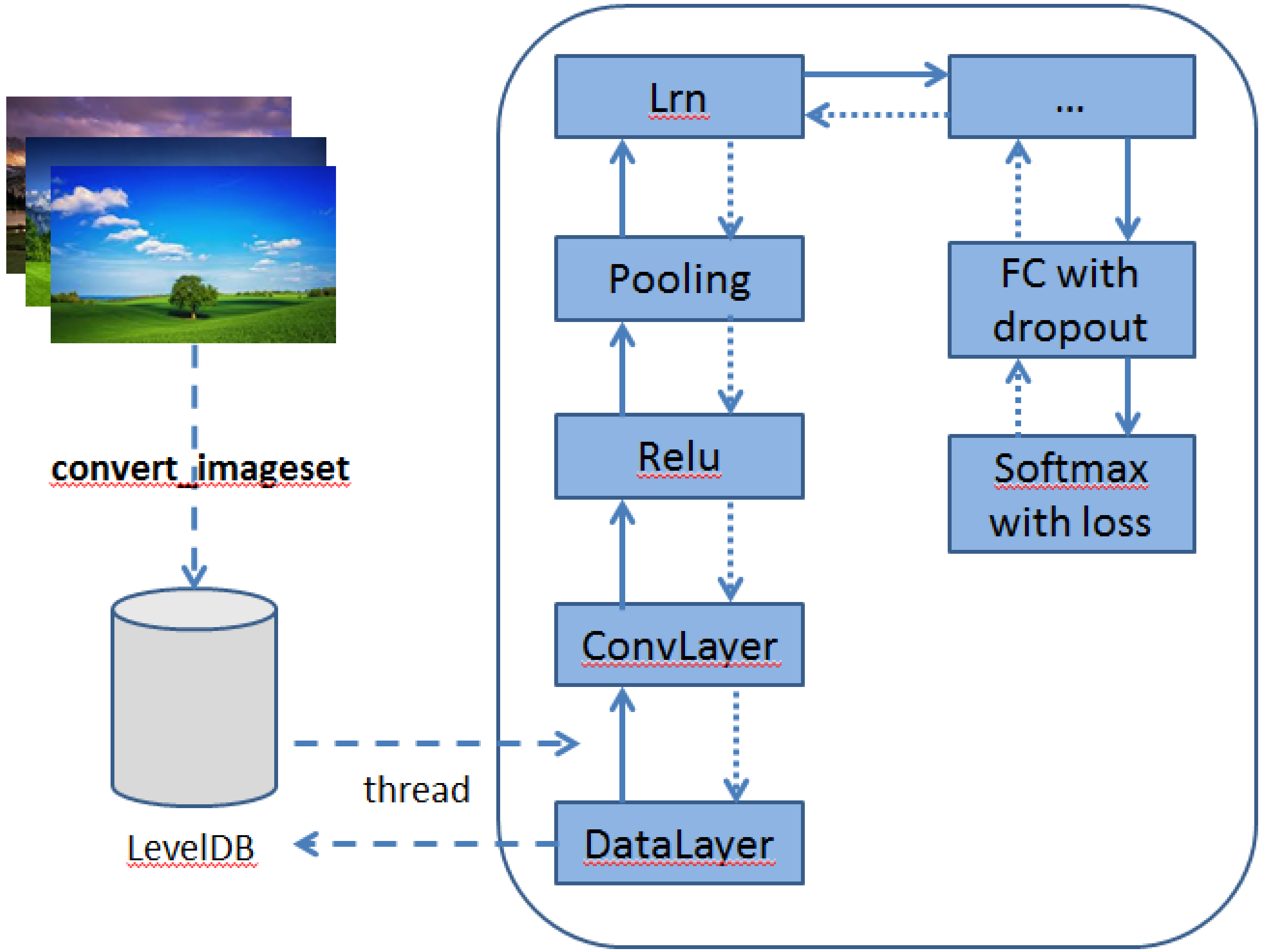
```
# cuDNN acceleration switch (uncomment to build with cuDNN).
USE_CUDNN := 1
CUDNN_PATH := $(ANACONDA_HOME)/pkgs/cudnn-5.1.0-1

# CUDA
CUDA_DIR := /usr/local/cuda-8.0

# CPU-only switch (uncomment to build without GPU support).
CPU_ONLY := 1

....
....
```

Essentials of Caffe

- Architecture
- Nets
- Layers
- Solver

Essentials of Caffe

- Architecture



LevelDB is a fast key-value storage library written at Google that provides an ordered mapping from string keys to string values.

Essentials of Caffe

Nets

Caffe defines a net layer-by-layer in its own model schema. The network defines the entire model bottom-to-top from input data to loss. As data and derivatives flow through the network in the forward and backward passes Caffe stores, communicates, and manipulates the information as blobs:the blob is the standard array and unified memory interface for the framework.

Essentials of Caffe

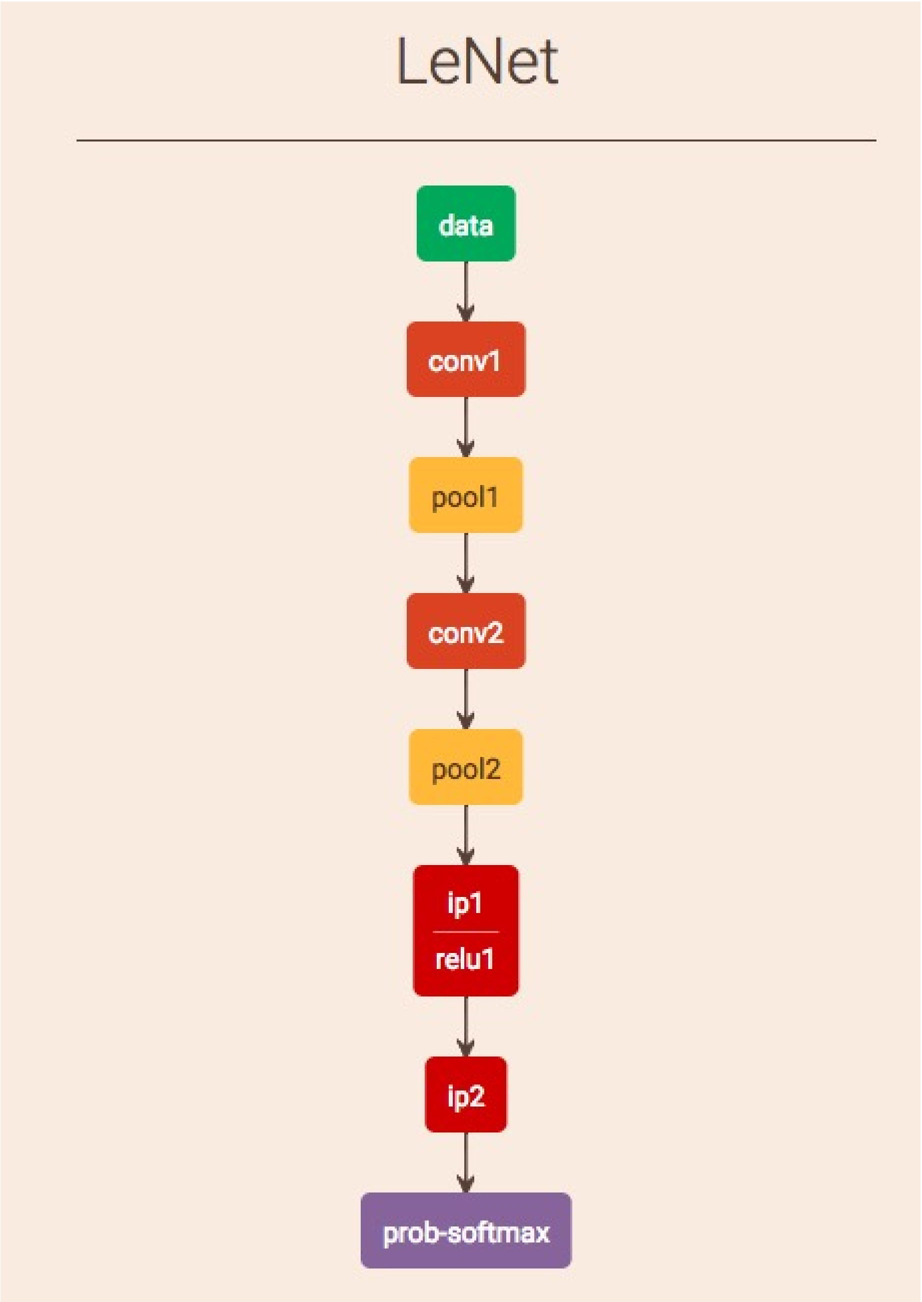
Nets

- Defined in a file with suffix: `.prototxt`, eg. `$caffe_root/caffe/examples/mnist/lenet.prototxt` defines a LeNet.
- Schema:

```
name: "dummy-net"
layers {name: "data" …}
layers {name: "conv" …}
layers {name: "pool" …}
layers {name: "loss" …}
```

Essentials of Caffe

Visualization of LeNet:



Essentials of Caffe

Layer

To create a Caffe model you need to define the model architecture in a protocol buffer definition file (prototxt).

- Data Layers
- Common Layers
- Vision Layers
- Recurrent Layers
- Normalization Layers
- Activation / Neuron Layers
- Loss Layers

Data 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.

Common layers

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

Vision Layers

Vision layers usually take images as input and produce other images as output, although they can take data of other types and dimensions.

- 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.
- Im2Col - relic helper layer that is not used much anymore.

Recurrent Layers

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

Normalization 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.

Activation / Neuron Layers

In general, activation / Neuron layers are element-wise operators, taking one bottom blob and producing one top blob of the same size.

- ReLU / Rectified-Linear and Leaky-ReLU - ReLU and Leaky-ReLU rectification.
- PReLU - parametric ReLU.
- ELU - exponential linear rectification.
- Sigmoid
- TanH
- ...

Loss 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
- 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.

Essentials of Caffe

Layer:

Input Data layer:

```
name: "LeNet"
layer {
  name: "data"
  type: "Input"
  top: "data"
  input_param { shape: { dim: 64 dim: 1 dim: 28 dim: 28 } }
}
```

Essentials of Caffe

Layer:

Convolution Layer:

```
layer {
  name: "conv1"
  type: "Convolution"
  bottom: "data"
  top: "conv1"
  convolution_param {
    num_output: 20
    kernel_size: 5
    stride: 1
    weight_filler {
      type: "xavier"
    }
    bias_filler {
      type: "constant"
    }
  }
}
```

Essentials of Caffe

Layer:

Pooling layer:

```
layer {
  name: "pool1"
  type: "Pooling"
  bottom: "conv1"
  top: "pool1"
  pooling_param {
    pool: MAX
    kernel_size: 2
    stride: 2
  }
```

Solver

The solver orchestrates model optimization by coordinating the network’s forward inference and backward gradients to form parameter updates that attempt to improve the loss.The responsibilities of learning are divided between the Solver for overseeing the optimization and generating parameter updates and the Net for yielding loss and gradients.

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")

```
-rwxr-xr-x@ 1 777 Nov 10 07:44 lenet_adadelta_solver.prototxt
-rwxr-xr-x@ 1 778 Nov 10 07:44 lenet_auto_solver.prototxt
-rwxr-xr-x@ 1 6003 Nov 10 07:44 lenet_consolidated_solver.proto
-rwxr-xr-x@ 1 871 Nov 10 07:44 lenet_multistep_solver.prototxt
-rwxr-xr-x@ 1 90 Nov 10 07:44 lenet_solver.prototxt
-rwxr-xr-x@ 1 886 Nov 10 07:44 lenet_solver_adam.prototxt
-rwxr-xr-x@ 1 830 Nov 10 07:44 lenet_solver_rmsprop.prototxt
```

`lenet_solver_rmsprop.prototxt`:

```
# The train/test net protocol buffer definition
test_iter: 100
# Carry out testing every 500 training iterations.
test_interval: 500
# The base learning rate, momentum and the weight decay of the
base_lr: 0.01
momentum: 0.0
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
# snapshot intermediate results
snapshot: 5000
snapshot_prefix: "examples/mnist/lenet_rmsprop"
# solver mode: CPU or GPU
solver_mode: GPU
type: "RMSProp"
rms_decay: 0.98
```

How to run?

Once you have your Solver and Net ready, you can start train you deep neural network by:

```
caffe train -solver examples/mnist/lenet_solver.prototxt -gpu 0,1
```

Deep Transfer Learning on Caffe

This caffe library for deep transfer learning is modified from Caffe(repository with version ID `29cdee7`) with the following modifications:

- Add `mmd layer` described in paper "Learning Transferable Features with Deep Adaptation Networks" (ICML '15).
- Add `jmmd layer` described in paper "Deep Transfer Learning with Joint Adaptation Networks" (ICML '17).

- Add `entropy layer` and `outerproduct layer` described in paper "Unsupervised Domain Adaptation with Residual Transfer Networks" (NIPS '16).

- Copy `grl layer` and `messenger.hpp` from repository Caffe.

- Emit `SOLVER_ITER_CHANGE` message in `solver.cpp` when `iter_` changes.

Data Preparation

In `data/office/*.txt`, we give the lists of three domains in Office dataset.

```
-rw-rw-r-- 1 fanyy fanyy 246760 Nov 22 19:54 amazon_list.txt
-rw-rw-r-- 1 fanyy fanyy  42744 Nov 22 19:54 dslr_list.txt
-rw-rw-r-- 1 fanyy fanyy  69666 Nov 22 19:54 webcam_list.txt
```

How to train?

Alexnet:

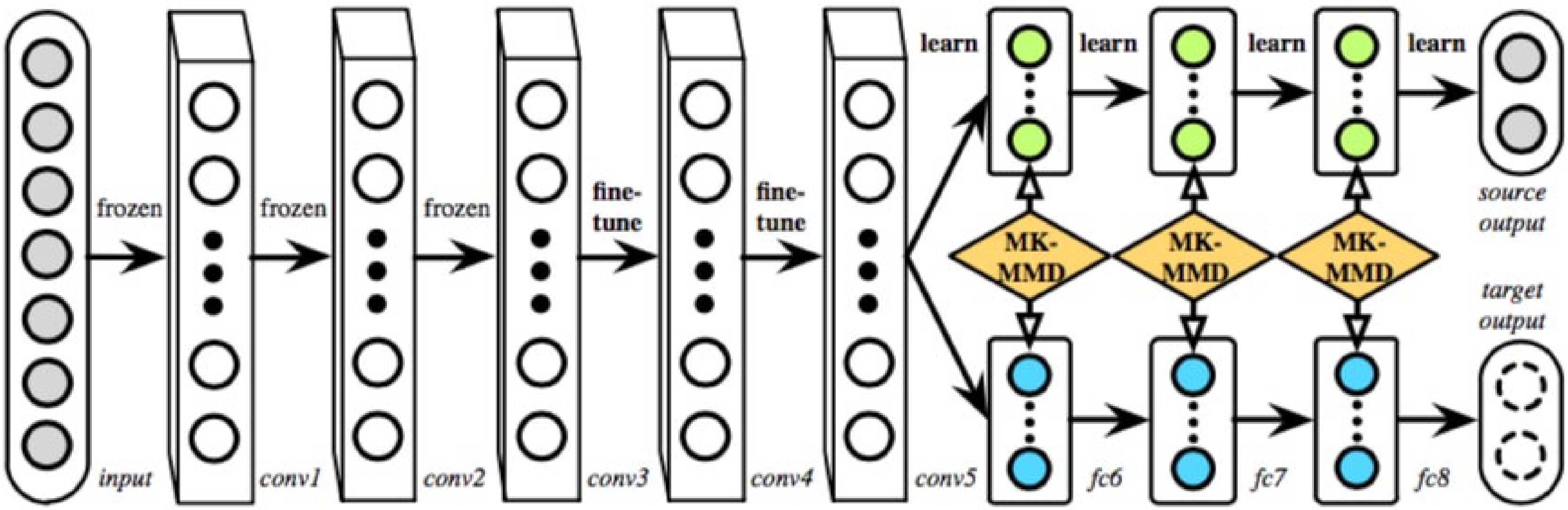
```
"./build/tools/caffe train -solver
models/*/alexnet/solver.prototxt -weights
models/bvlc_reference_caffenet/bvlc_reference_caffenet.caffemod
el (*=DAN, RTN or JAN)"
```

Training Model

- In `models/DAN/alexnet`, we give an example model based on Alexnet to show how to transfer from `amazon` to `webcam`.
  In this model, we insert mmd layers after fc7 and fc8 individually.
- In `models/JAN/alexnet`, we give an example model based on Alexnet to show how to transfer from `amazon` to `webcam`.
  In this model, we insert jmmd layers with outputs of fc7 and fc8 as its input.

Model Architecture

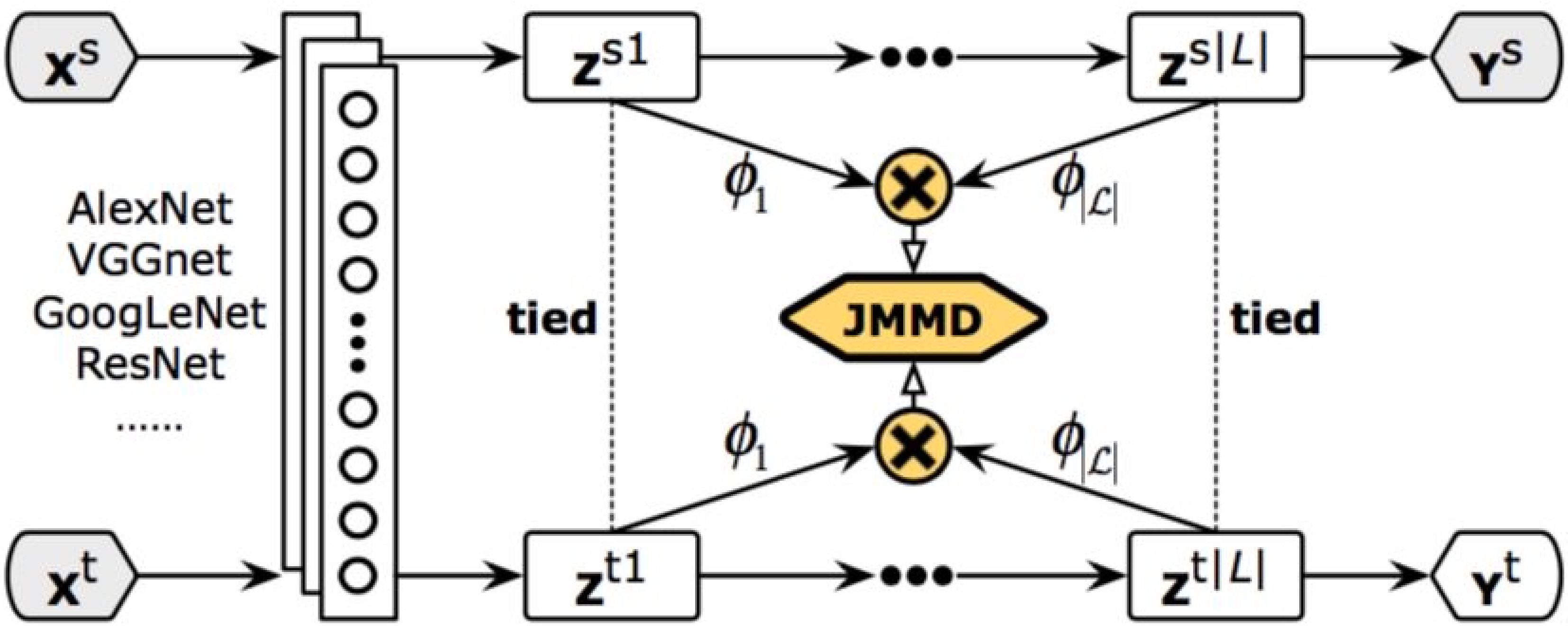
DAN(Deep Adaptation Network):



Defined in: `$caffe_root/models/DAN/alexnet/train_val.prototxt`

Model Architecture

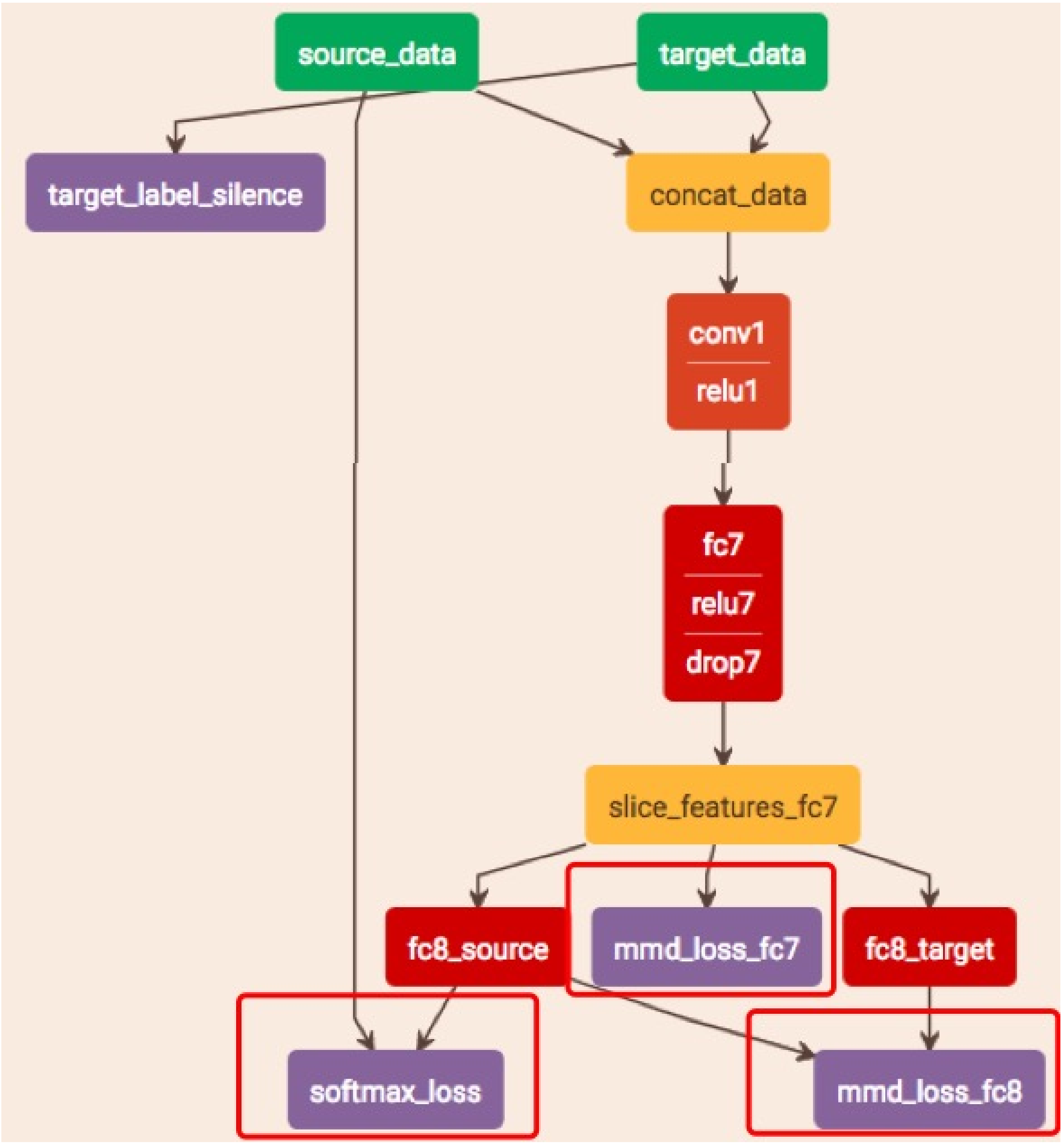
JAN(Joint Adaptation Networks):



(a) Joint Adaptation Network (JAN)

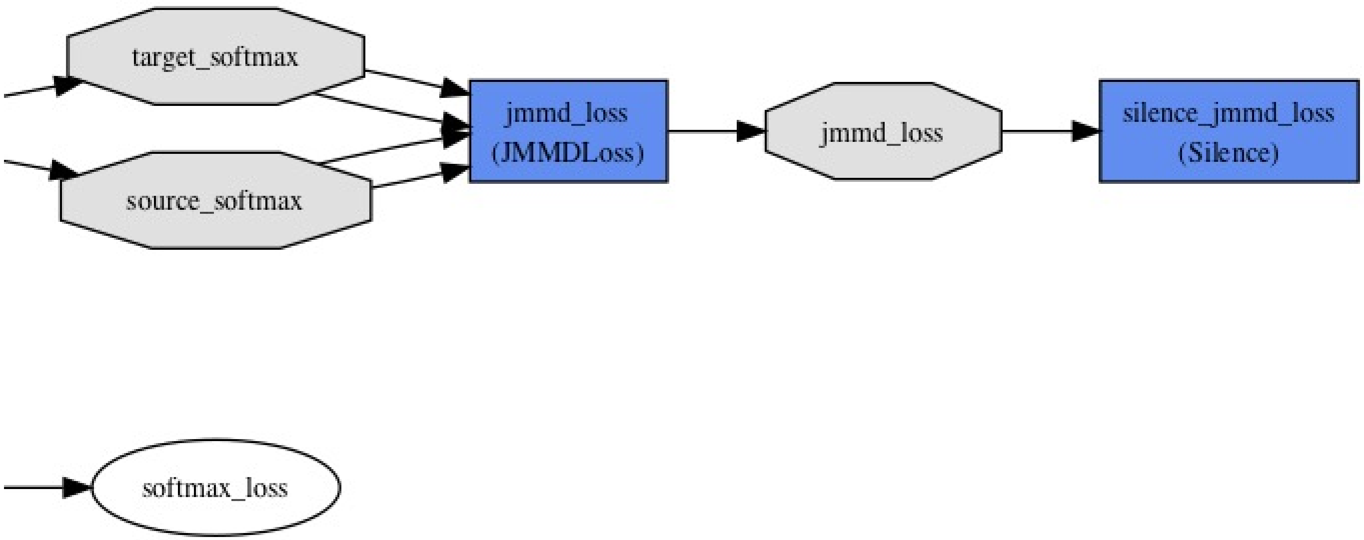
Defined in: `$caffe_root/models/JAN/alexnet/train_val.prototxt`

Visualization of DAN



MMD

Visualization of JAN



JMMD

Dive into the codes

General design of layers

- Interface:

  `src/caffe/layers/mmd_layer.cpp`

  `src/caffe/layers/jmmd_layer.cpp`

- Implementation:

  `src/caffe/layers/mmd_layer.cu`

  `src/caffe/layers/jmmd_layer.cu`

Dive into the codes

- Interface: `mmd_layer.cpp`:

```
void MMDLossLayer<Dtype>::LayerSetUp(
    const vector<Blob<Dtype>*>& bottom, const vector<Blob<Dtype
    }

void MMDLossLayer<Dtype>::Reshape(
    const vector<Blob<Dtype>*>& bottom, const vector<Blob<Dtype
    }

void MMDLossLayer<Dtype>::Forward_cpu(const vector<Blob<Dtype>*
    const vector<Blob<Dtype>*>& top) {
}

void MMDLossLayer<Dtype>::Backward_cpu(const vector<Blob<Dtype>
    const vector<bool>& propagate_down,
    const vector<Blob<Dtype>*>& bottom) {
}
```

Dive into the codes

- Implementation: `mmd_layer.cu`:

```
__global__ void CalculateKernel(const int n, const Dtype* dista
        Dtype* out) {
        ...
        }

__global__ void CalculateSpreadDistance2(const int n, const Dty
        const int source_num, const int target_num, const int c
        ...
        }
void calculate_diff(...) {
        ...
        }

void MMDLossLayer<Dtype>::Forward_gpu(const vector<Blob<Dtype>*
    const vector<Blob<Dtype>*>& top) {

void MMDLossLayer<Dtype>::Backward_gpu(const vector<Blob<Dtype>
    const vector<bool>& propagate_down,
    const vector<Blob<Dtype>*>& bottom) {
    ...
    }
```

Dive into the codes

Optimization Objective of DAN:

$$\mathrm{min}_\Theta \frac{1}{n_a}\sum_{i=1}^{n_a} J(\theta(X_i^a), y_i^a) + \lambda \sum_{l=l_1}^{l_2} d_k^2(D_s^l, D_t^l)$$

- Minmize cross-entropy loss function:
- MK-MMD-based multi-layer adaptation regularizer：

Dive into the codes

Implementation of MK-MMD

$$\sum_{l=l_1}^{l_2} d_k^2(D_s^l, D_t^l)$$

Implementation of MK-MMD:

d_k^2 is the square distance of a specified kernel function k:

```
__global__ void CalculateSpreadDistance2(const int n, const Dty
        const int source_num, const int target_num, const int d
  CUDA_KERNEL_LOOP(index, n) {
      int data_index1 = index / dim / (source_num + target_num)
      int data_index2 = index / dim % (source_num + target_num)
      int dim_offset = index % dim;
      Dtype data1;
      Dtype data2;
      if(data_index1 >= source_num){
          data_index1 -= source_num;
          data1 = target[data_index1 * dim + dim_offset];
      }else{
          data1 = source[data_index1 * dim + dim_offset];
      }
      if(data_index2 >= source_num){
          data_index2 -= source_num;
          data2 = target[data_index2 * dim + dim_offset];
      }else{
          data2 = source[data_index2 * dim + dim_offset];
      }
      out[index] = (data1 - data2) * (data1 - data2);
  }
```

Implementation of MK-MMD:

Kernel function k:

```
__global__ void CalculateKernel(const int n, const Dtype* dista
        Dtype* out) {
  CUDA_KERNEL_LOOP(index, n) {
      out[index] = exp(-gamma * distance2[index]);
  }
}
```

Implementation of MK-MMD:

Calculate each kernel of data:

```
//calculate each kernel of data
Dtype gamma_times = pow(kernel_mul_, (Dtype)(kernel_num_ / 2));
Dtype kernel_gamma = gamma_ / gamma_times;

nthreads = total_num_ * total_num_;
for(int i = 0;i < kernel_num_;++i){
        CalculateKernel<Dtype><<<CAFFE_GET_BLOCKS(nthreads), CA
        nthreads, distance2, kernel_gamma, kernel_val_[i]->muta
        kernel_gamma *= kernel_mul_;
}
```

What makes a kernel function different?

we choose different kernel_gamma!!!

Implementation of MK-MMD:

MK-MMD Loss:

```
Dtype loss = 0;
int sample_num = (source_num_ > target_num_) ? source_num_
int s1, s2, t1, t2;
for(int i = 0;i < sample_num;++i){
    s1 = rand() % source_num_;
    s2 = rand() % source_num_;
    s2 = (s1 != s2) ? s2 : (s2 + 1) % source_num_;

    t1 = rand() % target_num_;
    t2 = rand() % target_num_;
    t2 = (t1 != t2) ? t2 : (t2 + 1) % target_num_;

    for(int i = 0;i < kernel_num_;++i){
        loss += kernel_val_[i]->cpu_data()[s1 * total_num_
        loss += kernel_val_[i]->cpu_data()[(source_num_ + t
        loss -= kernel_val_[i]->cpu_data()[s1 * total_num_
        loss -= kernel_val_[i]->cpu_data()[s2 * total_num_
    }
}
```

Implementation of MK-MMD:

MK-MMD Loss:

```
kv[] = kernel_val_[i]->cpu_data();
for(int i = 0;i < kernel_num_; ++i){
    loss += kv[s1 * total_num_ + s2];
    loss += kv[(source_num_+t1)*total_num_ + source_num_ + t2];
    loss -= kv[s1 * total_num_ + source_num_ + t2];
    loss -= kv[s2 * total_num_ + source_num_ + t1];
}
```

why?

Unbiased estimate of MK-MMD:

$$d_k^2(p,q) = \frac{2}{n_s}\sum_{i=1}^{n_s/2} g_k(z_i)$$

where

$$g_k(z_i) = k(X_{2i-1}^s, X_{2i}^s) + k(X_{2i-1}^t, X_{2i}^t) -$$

$$k(X_{2i-1}^s, X_{2i}^t) - k(X_{2i}^s, X_{2i-1}^t)$$

which can be computed with linear complexity.

Dive into the codes

Optimization Objective of JAN:

$$\min_f \frac{1}{n_s} \sum_{i=1}^{n_s} J(f(X_i^s), y_s^s) + \lambda \hat{D}_L(P, Q)$$

where

$$\begin{aligned}\widehat{D}_{\mathcal{L}}(P,Q) = \frac{2}{n}\sum_{i=1}^{n/2}\left(\prod_{\ell\in\mathcal{L}} k^\ell(\mathbf{z}_{2i-1}^{s\ell},\mathbf{z}_{2i}^{s\ell}) + \prod_{\ell\in\mathcal{L}} k^\ell(\mathbf{z}_{2i-1}^{t\ell},\mathbf{z}_{2i}^{t\ell})\right) \\ - \frac{2}{n}\sum_{i=1}^{n/2}\left(\prod_{\ell\in\mathcal{L}} k^\ell(\mathbf{z}_{2i-1}^{s\ell},\mathbf{z}_{2i}^{t\ell}) + \prod_{\ell\in\mathcal{L}} k^\ell(\mathbf{z}_{2i-1}^{t\ell},\mathbf{z}_{2i}^{s\ell})\right),\end{aligned}$$

Implementation of JMMD:

- calculate square distance between each data pair -> distance2(CalculateElewiseSquareDistance)
- calculate bandwith of RBF kernel -> gamma_
- calculate each kernel of data
- calculate each kernel of label(CalculateLabelProbRBFKernel)

The main difference between MMD and JMMD is that: JMMD calculates the kernel of both data and label.

For details, please go to `jmmd_layer.cu`.

Thank you.