CS231n Caffe Tutorial
Outline

● Caffe walkthrough
● Finetuning example
  ○ With demo!
● Python interface
  ○ With demo!
Caffe
Most important tip...

Don’t be afraid to read the code!
Caffe: Main classes

- **Blob**: Stores data and derivatives (header source)
- **Layer**: Transforms bottom blobs to top blobs (header + source)
- **Net**: Many layers; computes gradients via forward / backward (header source)
- **Solver**: Uses gradients to update weights (header source)
Protocol Buffers

● Like strongly typed, binary JSON
● Developed by Google
● Define **message types** in `.proto` file
● Define **messages** in `.prototxt` or `.binaryproto` files (Caffe also uses `.caffemodel`)
● All Caffe messages defined [here](#):
  ○ This is a very important file!
Prototxt: Define Net

name: "LogisticRegressionNet"
layers {
  top: "data"
  top: "label"
  name: "data"
  type: HDF5_DATA
  hdf5_data_param {
    source: "examples/hdf5_classification/data/train.txt"
    batch_size: 10
  }
  include {
    phase: TRAIN
  }
}
layers {
  bottom: "data"
  top: "fc1"
  name: "fc1"
  type: INNER_PRODUCT
  blobs_lr: 1
  blobs_lr: 2
  weight_decay: 1
  weight_decay: 0
  inner_product_param {
    num_output: 2
    weight_filler {
      type: "gaussian"
      std: 0.01
    }
    bias_filler {
      type: "constant"
      value: 0
    }
  }
}
layers {
  bottom: "fc1"
  bottom: "label"
  top: "loss"
  name: "loss"
  type: SOFTMAX_LOSS
}
Prototxt: Define Net

name: "LogisticRegressionNet"
layers {
  top: "data"
  top: "label"
  name: "data"
  type: HDF5_DATA
  hdf5_data_param {
    source: "examples/hdf5_classification/data/train.txt"
    batch_size: 10
  }
  include {
    phase: TRAIN
  }
}
layers {
  bottom: "data"
  top: "fc1"
  name: "fc1"
  type: INNER_PRODUCT
  blobs_lr: 1
  blobs_lr: 2
  weight_decay: 1
  weight_decay: 0
}
inner_product_param {
  num_output: 2
  weight_filler {
    type: "gaussian"
    std: 0.01
  }
  bias_filler {
    type: "constant"
    value: 0
  }
}
layers {
  bottom: "fc1"
  bottom: "label"
  top: "loss"
  name: "loss"
  type: SOFTMAXLOSS
}
Prototxt: Define Net

```protobuf
define_net {
  name: "LogisticRegressionNet"
  layers {
    top: "data"
    top: "label"
    name: "data"
    type: HDF5_DATA
    hdf5_data_param {
      source: "examples/hdf5_classification/data/train.txt"
      batch_size: 10
    }
    include {
      phase: TRAIN
    }
  }
  layers {
    bottom: "data"
    top: "fc1"
    name: "fc1"
    type: INNER_PRODUCT
    blobs_lr: 1
    blobs_lr: 2
    weight_decay: 1
    weight_decay: 0
  }
  inner_product_param {
    num_output: 2
    weight_filler {
      type: "gaussian"
      std: 0.01
    }
    bias_filler {
      type: "constant"
      value: 0
    }
  }
  layers {
    bottom: "fc1"
    bottom: "label"
    top: "loss"
    name: "loss"
    type: SOFTMAX_LOSS
  }
}
```

Layers and Blobs often have same name!

Learning rates (weight + bias)

Regularization (weight + bias)
Prototxt: Define Net

Layers and Blobs often have same name!

Learning rates (weight + bias)

Regularization (weight + bias)

Number of output classes
Prototxt: Define Net

Layers and Blobs often have same name!

Set these to 0 to freeze a layer

Learning rates (weight + bias)

Regularization (weight + bias)

Number of output classes

```
name: "LogisticRegressionNet"
layers {
top: "data"
top: "label"
name: "data"
type: HDF5 DATA
hdf5_data_param {
  source: "examples/hdf5_classification/data/train.txt"
batch_size: 10
}
include {
  phase: TRAIN
}
layers {
  bottom: "data"
top: "fc1"
name: "fc1"
type: INNER PRODUCT
blobs_lr: 1
blobs_lr: 2
weight_decay: 1
weight_decay: 0
inner_product_param {
  num_output: 2
  weight_filler {
    type: "gaussian"
    std: 0.01
  }
bias_filler {
    type: "constant"
    value: 0
  }
}
```
Getting data in: DataLayer

- Reads images and labels from LMDB file
- Only good for 1-of-k classification
- Use this if possible
  - [header source proto](#)
Getting data in: DataLayer

layer {
  name: "data"
  type: "Data"
  top: "data"
  top: "label"
  include {
    phase: TRAIN
  }
  transform_param {
    mirror: true
    crop_size: 227
    mean_file: "data/ilsvrc12/imagenet_mean.binaryproto"
  }
  data_param {
    source: "examples/imagenet/ilsvrc12_train_lmdb"
    batch_size: 256
    backend: LMDB
  }
}
Getting data in: ImageDataLayer

- Get images and labels directly from image files
- No LMDB but probably slower than DataLayer
- May be faster than DataLayer if reading over network? Try it out and see

(header source proto)
Getting data in: WindowDataLayer

- Read windows from image files and class labels
- Made for detection
  - (header source proto)
Getting data in: HDF5Layer

- Reads arbitrary data from HDF5 files
  - Easy to read / write in Python using h5py
- Good for any task - regression, etc
- Other DataLayers do prefetching in a separate thread, HDF5Layer does not
- Can only store float32 and float64 data - no uint8 means image data will be huge
- Use this if you have to
  - (header source proto)
Getting data in: from memory

- Manually copy data into the network
- Slow; don’t use this for training
- Useful for quickly visualizing results
- Example later
Data augmentation

- Happens on-the-fly!
  - Random crops
  - Random horizontal flips
  - Subtract mean image
- See TransformationParameter proto
- DataLayer, ImageDataLayer, WindowDataLayer
- NOT HDF5Layer
Finetuning
Basic Recipe

1. Convert data
2. Define net (as prototxt)
3. Define solver (as prototxt)
4. Train (with pretrained weights)
Convert Data

- DataLayer reading from LMDB is the easiest
- Create LMDB using `convert_imageset`
- Need text file where each line is
  - “[path/to/image.jpeg] [label]”
Define Net

- Write a `.prototxt` file defining a `NetParameter`
- If finetuning, copy existing `.prototxt` file
  - Change data layer
  - Change output layer: name and `num_output`
  - Reduce batch size if your GPU is small
  - Set `blobs_lr` to 0 to “freeze” layers
Define Solver

- Write a prototxt file defining a `SolverParameter`
- If finetuning, copy existing solver.prototxt file
  - Change net to be your net
  - Change snapshot_prefix to your output
  - Reduce base learning rate (divide by 100)
  - Maybe change max_iter and snapshot
Define net: Change layer name

Original prototxt:
layer {
    name: "fc7"
    type: "InnerProduct"
    inner_product_param {
        num_output: 4096
    }
}

[... ReLU, Dropout]
layer {
    name: "fc8"
    type: "InnerProduct"
    inner_product_param {
        num_output: 1000
    }
}

Pretrained weights:
"fc7.weight": [values]
"fc7.bias": [values]
"fc8.weight": [values]
"fc8.bias": [values]

Modified prototxt:
layer {
    name: "fc7"
    type: "InnerProduct"
    inner_product_param {
        num_output: 4096
    }
}

[... ReLU, Dropout]
layer {
    name: "my-fc8"
    type: "InnerProduct"
    inner_product_param {
        num_output: 10
    }
}
Define net: Change layer name

Original prototxt:
```
layer {
    name: "fc7"
    type: "InnerProduct"
    inner_product_param {
        num_output: 4096
    }
}

[... ReLU, Dropout]
layer {
    name: "fc8"
    type: "InnerProduct"
    inner_product_param {
        num_output: 1000
    }
}
```

Pretrained weights:
- fc7.weight: [values]
- fc7.bias: [values]
- fc8.weight: [values]
- fc8.bias: [values]

Modified prototxt:
```
layer {
    name: "fc7"
    type: "InnerProduct"
    inner_product_param {
        num_output: 4096
    }
}

[... ReLU, Dropout]
layer {
    name: "my-fc8"
    type: "InnerProduct"
    inner_product_param {
        num_output: 10
    }
}
```

Same name: weights copied

Pretrained weights:
- fc7.weight: [values]
- fc7.bias: [values]
Define net: Change layer name

Original prototxt:
layer {
    name: "fc7"
    type: "InnerProduct"
    inner_product_param {
        num_output: 4096
    }
}

[... ReLU, Dropout]
layer {
    name: "fc8"
    type: "InnerProduct"
    inner_product_param {
        num_output: 1000
    }
}

Modified prototxt:
layer {
    name: "fc7"
    type: "InnerProduct"
    inner_product_param {
        num_output: 4096
    }
}

[... ReLU, Dropout]
layer {
    name: "my-fc8"
    type: "InnerProduct"
    inner_product_param {
        num_output: 10
    }
}

Pretrained weights:
"fc7.weight": [values]
"fc7.bias": [values]
"fc8.weight": [values]
"fc8.bias": [values]

Different name:
weights reinitialized
Demo!

hopefully it works...
Python interface
Not much documentation...

Read the code! Two most important files:

- **caffe/python/caffe/__caffe.cpp**:
  - Exports Blob, Layer, Net, and Solver classes

- **caffe/python/caffe/pycaffe.py**
  - Adds extra methods to Net class
Python Blobs

- Exposes data and diffs as numpy arrays
- Manually feed data to the network by copying to input numpy arrays
Python Layers

- `layer.blobs` gives a list of `Blobs` for parameters of a layer
- It’s possible to define new types of layers in Python, but still experimental
  - (code unit test)
Python Nets

Some useful methods:

- **constructors**: Initialize Net from model prototxt file and (optionally) weights file
- **forward**: run forward pass to compute loss
- **backward**: run backward pass to compute derivatives
- **forward_all**: Run forward pass, batching if input data is bigger than net batch size
- **forward_backward_all**: Run forward and backward passes in batches
Python Solver

- Can replace *caffe train* and instead use Solver directly from Python
- Example in *unit test*
Net vs Classifier vs Detector ... ?

- Most important class is Net, but there are others
- **Classifier** (code main):
  - Extends Net to perform classification, averaging over 10 image crops
- **Detector** (code main):
  - Extends Net to perform R-CNN style detection
- Don’t use these, but read them to see how Net works
Model ensembles

- No built-in support; do it yourself
Questions?