Caffe

Deep learning framework by the BVLC

Created by
Yangqing Jia
Lead Developer
Evan Shelhamer

View On GitHub

Training LeNet on MNIST with Caffe

We will assume that you have Caffe successfully compiled. If not, please refer to the Installation page. In this tutorial, we will assume that your Caffe installation is located at `CAFFE_ROOT`.

Prepare Datasets

You will first need to download and convert the data format from the MNIST website. To do this, simply run the following commands:

```
cd $CAFFE_ROOT
./data/mnist/get_mnist.sh
./examples/mnist/create_mnist.sh
```

If it complains that `wget` or `gunzip` are not installed, you need to install them respectively. After running the script there should be two datasets, `mnist_train_lmdb`, and `mnist_test_lmdb`.

LeNet: the MNIST Classification Model

Before we actually run the training program, let's explain what will happen. We will use the LeNet network, which is known to work well on digit classification tasks. We will use a slightly different version from the original

LeNet implementation, replacing the sigmoid activations with Rectified Linear Unit (ReLU) activations for the neurons.

The design of LeNet contains the essence of CNNs that are still used in larger models such as the ones in ImageNet. In general, it consists of a convolutional layer followed by a pooling layer, another convolution layer followed by a pooling layer, and then two fully connected layers similar to the conventional multilayer perceptrons. We have defined the layers in `$CAFFE_ROOT/examples/mnist/lenet_train_test.prototxt`.

Define the MNIST Network

This section explains the `lenet_train_test.prototxt` model definition that specifies the LeNet model for MNIST handwritten digit classification. We assume that you are familiar with Google Protobuf, and assume that you have read the protobuf definitions used by Caffe, which can be found at `$CAFFE_ROOT/src/caffe/proto/caffe.proto`.

Specifically, we will write a `caffe::NetParameter` (or in python, `caffe.proto.caffe_pb2.NetParameter`) protobuf. We will start by giving the network a name:

```
name: "LeNet"
```

Writing the Data Layer

Currently, we will read the MNIST data from the lmdb we created earlier in the demo. This is defined by a data layer:

```
layer {
  name: "mnist"
  type: "Data"
  transform_param {
    scale: 0.00390625
  }
  data_param {
    source: "mnist_train_lmdb"
    backend: LMDB
    batch_size: 64
  }
  top: "data"
  top: "label"
}
```

Specifically, this layer has name `mnist`, type `data`, and it reads the data from the given lmdb source. We will use a batch size of 64, and scale the incoming

pixels so that they are in the range [0,1). Why 0.00390625? It is 1 divided by 256. And finally, this layer produces two blobs, one is the `data` blob, and one is the `label` blob.

Writing the Convolution Layer

Let's define the first convolution layer:

```
layer {
  name: "conv1"
  type: "Convolution"
  param { lr_mult: 1 }
  param { lr_mult: 2 }
  convolution_param {
    num_output: 20
    kernel_size: 5
    stride: 1
    weight_filler {
      type: "xavier"
    }
    bias_filler {
      type: "constant"
    }
  }
  bottom: "data"
  top: "conv1"
}
```

This layer takes the `data` blob (it is provided by the data layer), and produces the `conv1` layer. It produces outputs of 20 channels, with the convolutional kernel size 5 and carried out with stride 1.

The fillers allow us to randomly initialize the value of the weights and bias. For the weight filler, we will use the `xavier` algorithm that automatically determines the scale of initialization based on the number of input and output neurons. For the bias filler, we will simply initialize it as constant, with the default filling value 0.

`lr_mult`s are the learning rate adjustments for the layer's learnable parameters. In this case, we will 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 - this usually leads to better convergence rates.

Writing the Pooling Layer

Phew. Pooling layers are actually much easier to define:

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

This says we will perform max pooling with a pool kernel size 2 and a stride of 2 (so no overlapping between neighboring pooling regions).

Similarly, you can write up the second convolution and pooling layers. Check `$CAFFE_ROOT/examples/mnist/lenet_train_test.prototxt` for details.

Writing the Fully Connected Layer

Writing a fully connected layer is also simple:

```
layer {
  name: "ip1"
  type: "InnerProduct"
  param { lr_mult: 1 }
  param { lr_mult: 2 }
  inner_product_param {
    num_output: 500
    weight_filler {
      type: "xavier"
    }
    bias_filler {
      type: "constant"
    }
  }
  bottom: "pool2"
  top: "ip1"
}
```

This defines a fully connected layer (known in Caffe as an `InnerProduct` layer) with 500 outputs. All other lines look familiar, right?

Writing the ReLU Layer

A ReLU Layer is also simple:

```
layer {
  name: "relu1"
  type: "ReLU"
  bottom: "ip1"
  top: "ip1"
}
```

Since ReLU is an element-wise operation, we can do in-place operations to save some memory. This is achieved by simply giving the same name to the bottom and top blobs. Of course, do NOT use duplicated blob names for other layer types!

After the ReLU layer, we will write another innerproduct layer:

```
layer {
  name: "ip2"
  type: "InnerProduct"
  param { lr_mult: 1 }
  param { lr_mult: 2 }
  inner_product_param {
    num_output: 10
    weight_filler {
      type: "xavier"
    }
    bias_filler {
      type: "constant"
    }
  }
  bottom: "ip1"
  top: "ip2"
}
```

Writing the Loss Layer

Finally, we will write the loss!

```
layer {
  name: "loss"
  type: "SoftmaxWithLoss"
  bottom: "ip2"
  bottom: "label"
}
```

The `softmax_loss` layer implements both the softmax and the multinomial logistic loss (that saves time and improves numerical stability). It takes two blobs, the first one being the prediction and the second one being the `label` provided by the data layer (remember it?). It does not produce any outputs - all it does is to compute the loss function value, report it when backpropagation starts, and initiates the gradient with respect to `ip2`. This is where all magic starts.

Additional Notes: Writing Layer Rules

Layer definitions can include rules for whether and when they are included in the network definition, like the one below:

```
layer {
  // ...layer definition...
  include: { phase: TRAIN }
}
```

This is a rule, which controls layer inclusion in the network, based on current network's state. You can refer to `$CAFFE_ROOT/src/caffe/proto/caffe.proto` for more information about layer rules and model schema.

In the above example, this layer will be included only in `TRAIN` phase. If we change `TRAIN` with `TEST`, then this layer will be used only in test phase. By default, that is without layer rules, a layer is always included in the network. Thus, `lenet_train_test.prototxt` has two `DATA` layers defined (with different `batch_size`), one for the training phase and one for the testing phase. Also, there is an `Accuracy` layer which is included only in `TEST` phase for reporting the model accuracy every 100 iteration, as defined in `lenet_solver.prototxt`.

Define the MNIST Solver

Check out the comments explaining each line in the prototxt `$CAFFE_ROOT/examples/mnist/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
# snapshot intermediate results
snapshot: 5000
snapshot_prefix: "examples/mnist/lenet"
# solver mode: CPU or GPU
solver_mode: GPU
```

Training and Testing the Model

Training the model is simple after you have written the network definition

protobuf and solver protobuf files. Simply run `train_lenet.sh`, or the following command directly:

```
cd $CAFFE_ROOT
./examples/mnist/train_lenet.sh
```

`train_lenet.sh` is a simple script, but here is a quick explanation: the main tool for training is `caffe` with action `train` and the solver protobuf text file as its argument.

When you run the code, you will see a lot of messages flying by like this:

```
I1203 net.cpp:66] Creating Layer conv1
I1203 net.cpp:76] conv1 <- data
I1203 net.cpp:101] conv1 -> conv1
I1203 net.cpp:116] Top shape: 20 24 24
I1203 net.cpp:127] conv1 needs backward computation.
```

These messages tell you the details about each layer, its connections and its output shape, which may be helpful in debugging. After the initialization, the training will start:

```
I1203 net.cpp:142] Network initialization done.
I1203 solver.cpp:36] Solver scaffolding done.
I1203 solver.cpp:44] Solving LeNet
```

Based on the solver setting, we will print the training loss function every 100 iterations, and test the network every 1000 iterations. You will see messages like this:

```
I1203 solver.cpp:204] Iteration 100, lr = 0.00992565
I1203 solver.cpp:66] Iteration 100, loss = 0.26044
...
I1203 solver.cpp:84] Testing net
I1203 solver.cpp:111] Test score #0: 0.9785
I1203 solver.cpp:111] Test score #1: 0.0606671
```

For each training iteration, `lr` is the learning rate of that iteration, and `loss` is the training function. For the output of the testing phase, score 0 is the accuracy, and score 1 is the testing loss function.

And after a few minutes, you are done!

```
I1203 solver.cpp:84] Testing net
I1203 solver.cpp:111] Test score #0: 0.9897
I1203 solver.cpp:111] Test score #1: 0.0324599
I1203 solver.cpp:126] Snapshotting to lenet_iter_10000
I1203 solver.cpp:133] Snapshotting solver state to lenet_iter_10000.solverstate
I1203 solver.cpp:78] Optimization Done.
```

The final model, stored as a binary protobuf file, is stored at

```
lenet_iter_10000
```

which you can deploy as a trained model in your application, if you are training on a real-world application dataset.

Um… How about GPU training?

You just did! All the training was carried out on the GPU. In fact, if you would like to do training on CPU, you can simply change one line in `lenet_solver.prototxt`:

```
# solver mode: CPU or GPU
solver_mode: CPU
```

and you will be using CPU for training. Isn't that easy?

MNIST is a small dataset, so training with GPU does not really introduce too much benefit due to communication overheads. On larger datasets with more complex models, such as ImageNet, the computation speed difference will be more significant.

How to reduce the learning rate at fixed steps?

Look at lenet_multistep_solver.prototxt