caffe.berkeleyvision.org

Caffe | Solver / Model Optimization

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"`)

The solver

1. scaffolds the optimization bookkeeping and creates the training network for learning and test network(s) for evaluation.
2. iteratively optimizes by calling forward / backward and updating parameters
3. (periodically) evaluates the test networks
4. snapshots the model and solver state throughout the optimization

where each iteration

1. calls network forward to compute the output and loss
2. calls network backward to compute the gradients
3. incorporates the gradients into parameter updates according to the solver method
4. updates the solver state according to learning rate, history, and method

to take the weights all the way from initialization to learned model.

Like Caffe models, Caffe solvers run in CPU / GPU modes.

Methods

The solver methods address the general optimization problem of loss minimization. For dataset D D, the optimization objective is the average loss over all |D||D| data instances throughout the dataset

L(W)=1|D|∑i|D|fW(X(i))+λr(W)

$$L(W) = \frac{1}{|D|}\sum_{i}^{|D|} f_W\left(X^{(i)}\right) + \lambda r(W)$$

where fW(X(i)) $f_W\left(X^{(i)}\right)$ is the loss on data instance X(i) $X^{(i)}$ and r(W) $r(W)$ is a regularization term with weight λ λ. |D||D| can be very large, so in practice, in each solver iteration we use a stochastic approximation of this objective, drawing a mini-batch of N<<|D| $N << |D|$ instances:

L(W)≈1N∑iNfW(X(i))+λr(W)

$$L(W) \approx \frac{1}{N}\sum_{i}^{N} f_W\left(X^{(i)}\right) + \lambda r(W)$$

The model computes fW f_W in the forward pass and the gradient ∇fW ∇f_W in the backward pass.

The parameter update ΔW ΔW is formed by the solver from the error gradient ∇fW ∇f_W, the regularization gradient ∇r(W) $\nabla r(W)$, and other particulars to each method.

SGD

Stochastic gradient descent (`type: "SGD"`) updates the weights WW by a linear combination of the negative gradient ∇L(W)$\nabla L(W)$ and the previous weight update VtV_t. The **learning rate** αα is the weight of the negative gradient. The **momentum** μμ is the weight of the previous update.

Formally, we have the following formulas to compute the update value Vt+1V_{t+1} and the updated weights Wt+1W_{t+1} at iteration t+1$t+1$, given the previous weight update VtV_t and current weights WtW_t:

Vt+1=μVt−α∇L(Wt)

$$V_{t+1} = \mu V_t - \alpha \nabla L(W_t)$$

Wt+1=Wt+Vt+1

$$W_{t+1} = W_t + V_{t+1}$$

The learning "hyperparameters" (αα and μμ) might require a bit of tuning for best results. If you're not sure where to start, take a look at the "Rules of thumb" below, and for further information you might refer to Leon Bottou's Stochastic Gradient Descent Tricks [1].

[1] L. Bottou. Stochastic Gradient Descent Tricks. *Neural Networks: Tricks of the Trade*: Springer, 2012.

Rules of thumb for setting the learning rate αα and momentum μμ

A good strategy for deep learning with SGD is to initialize the learning rate αα to a value around α≈0.01=10−2$\alpha \approx 0.01 = 10^{-2}$, and dropping it by a constant factor (e.g., 10) throughout training when the loss begins to reach an apparent "plateau", repeating this several times. Generally, you probably want to use a momentum μ=0.9$\mu = 0.9$ or similar value. By smoothing the weight updates across iterations, momentum tends to make deep learning with SGD both stabler and faster.

This was the strategy used by Krizhevsky et al. [1] in their famously winning CNN entry to the ILSVRC-2012 competition, and Caffe makes this strategy easy to implement in a `SolverParameter` , as in our reproduction of [1] at `./examples/imagenet/alexnet_solver.prototxt` .

To use a learning rate policy like this, you can put the following lines somewhere in your solver prototxt file:

```
 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 10
                  # (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

momentum: 0.9
```

Under the above settings, we'll always use `momentum` $\mu=0.9$. We'll begin training at a `base_lr` of $\alpha=0.01=10^{-2}$ for the first 100,000 iterations, then multiply the learning rate by `gamma` (γ) and train at $\alpha'=\alpha\gamma=(0.01)(0.1)=0.001=10^{-3}$ for iterations 100K-200K, then at $\alpha''=10^{-4}$ for iterations 200K-300K, and finally train until iteration 350K (since we have `max_iter: 350000`) at $\alpha'''=10^{-5}$.

Note that the momentum setting μ effectively multiplies the size of your updates by a factor of $\frac{1}{1-\mu}$ after many iterations of training, so if you increase μ, it may be a good idea to **decrease** α accordingly (and vice versa).

For example, with $\mu=0.9$, we have an effective update size multiplier of $10=\frac{1}{1-0.9}$. If we increased the momentum to $\mu=0.99$, we've increased our update size multiplier to 100, so we should drop α (`base_lr`) by a factor of 10.

Note also that the above settings are merely guidelines, and they're definitely not guaranteed to be optimal (or even work at all!) in every situation. If learning diverges (e.g., you start to see very large or `NaN` or `inf` loss values or outputs), try dropping the `base_lr` (e.g., `base_lr: 0.001`) and re-training, repeating this until you find a `base_lr` value that works.

[1] A. Krizhevsky, I. Sutskever, and G. Hinton. ImageNet Classification with Deep Convolutional Neural Networks. *Advances in Neural Information Processing Systems*, 2012.

AdaDelta

The **AdaDelta** (`type: "AdaDelta"`) method (M. Zeiler [1]) is a “robust learning rate method”. It is a gradient-based optimization method (like SGD). The update formulas are

(vt)iRMS(∇L(Wt))iE[g2]t=RMS((vt−1)i)RMS(∇L(Wt))i(∇L(Wt′))i=E[g2]+ε−−−−−−−√=δE[g2]t−1+(1−δ)g2t

$$(v_t)_i = \frac{\text{RMS}((v_{t-1})_i)}{\text{RMS}\left(\nabla L(W_t)\right)_i}\left(\nabla L(W_{t'})\right)_i$$

$$\text{RMS}\left(\nabla L(W_t)\right)_i = \sqrt{E[g^2] + \varepsilon}$$

$$E[g^2]_t = \delta E[g^2]_{t-1} + (1-\delta) g_t^2$$

and

(Wt+1)i=(Wt)i−α(vt)i.

$$(W_{t+1})_i = (W_t)_i - \alpha (v_t)_i .$$

[1] M. Zeiler ADADELTA: AN ADAPTIVE LEARNING RATE METHOD. *arXiv preprint*, 2012.

AdaGrad

The **adaptive gradient** (`type: "AdaGrad"`) method (Duchi et al. [1]) is a gradient-based optimization method (like SGD) that attempts to “find needles in haystacks in the form of very predictive but rarely seen features,” in Duchi et al.’s words. Given the update information from all previous iterations (∇L(W))t′(∇ $L(W))_{t'}$ for t′∈{1,2,...,t}$t' \in \{1, 2, \ldots, t\}$, the update formulas proposed by [1] are as follows, specified for each component ii of the weights WW:

(Wt+1)i=(Wt)i−α(∇L(Wt))i∑tt′=1(∇L(Wt′))2i−−−−−−−−−−−−−−√

$$(W_{t+1})_i = (W_t)_i - \alpha \frac{\left(\nabla L(W_t)\right)_i}{\sqrt{\sum_{t'=1}^{t} \left(\nabla L(W_{t'})\right)_i^2}}$$

Note that in practice, for weights W∈Rd$W \in \mathcal{R}^d$, AdaGrad implementations (including the one in Caffe) use only O(d)$\mathcal{O}(d)$ extra storage for the historical gradient information (rather than the O(dt)$\mathcal{O}(dt)$ storage that would be necessary to store each historical gradient individually).

[1] J. Duchi, E. Hazan, and Y. Singer. Adaptive Subgradient Methods for Online Learning and

Stochastic Optimization. *The Journal of Machine Learning Research*, 2011.

Adam

The **Adam** (`type: "Adam"`), proposed in Kingma et al. [1], is a gradient-based optimization method (like SGD). This includes an "adaptive moment estimation" (mt,vtm_t, v_t) and can be regarded as a generalization of AdaGrad. The update formulas are

(mt)i=β1(mt−1)i+(1−β1)(∇L(Wt))i,(vt)i=β2(vt−1)i+(1−β2)(∇L(Wt))2i
$(m_t)_i = \beta_1 (m_{t-1})_i + (1-\beta_1)(\nabla L(W_t))_i, (v_t)_i = \beta_2 (v_{t-1})_i + (1-\beta_2)(\nabla L(W_t))_i^2$

and

(Wt+1)i=(Wt)i−α1−(β2)ti−−−−−−−√1−(β1)ti(mt)i(vt)i−−−−√+ε.
$$(W_{t+1})_i = (W_t)_i - \alpha \frac{\sqrt{1-(\beta_2)_i^t}}{1-(\beta_1)_i^t} \frac{(m_t)_i}{\sqrt{(v_t)_i} + \varepsilon}.$$

Kingma et al. [1] proposed to use β1=0.9,β2=0.999,ε=10−8$\beta_1 = 0.9, \beta_2 = 0.999, \varepsilon = 10^{-8}$ as default values. Caffe uses the values of `momemtum, momentum2, delta` for β1,β2,ε $\beta_1, \beta_2, \varepsilon$, respectively.

[1] D. Kingma, J. Ba. Adam: A Method for Stochastic Optimization. *International Conference for Learning Representations*, 2015.

NAG

Nesterov's accelerated gradient (`type: "Nesterov"`) was proposed by Nesterov [1] as an "optimal" method of convex optimization, achieving a convergence rate of O(1/t2)$\mathcal{O}(1/t^2)$ rather than the O(1/t)$\mathcal{O}(1/t)$. Though the required assumptions to achieve the O(1/t2)$\mathcal{O}(1/t^2)$ convergence typically will not hold for deep networks trained with Caffe (e.g., due to non-smoothness and non-convexity), in practice NAG can be a very effective method for optimizing certain types of deep learning architectures, as demonstrated for deep MNIST autoencoders by Sutskever et al. [2].

The weight update formulas look very similar to the SGD updates given above:

Vt+1=μVt−α∇L(Wt+μVt)
$$V_{t+1} = \mu V_t - \alpha \nabla L(W_t + \mu V_t)$$

Wt+1=Wt+Vt+1

$$W_{t+1} = W_t + V_{t+1}$$

What distinguishes the method from SGD is the weight setting WW on which we compute the error gradient ∇L(W)$\nabla L(W)$ – in NAG we take the gradient on weights with added momentum ∇L(Wt+μVt)$\nabla L(W_t + \mu V_t)$; in SGD we simply take the gradient ∇L(Wt)$\nabla L(W_t)$ on the current weights themselves.

[1] Y. Nesterov. A Method of Solving a Convex Programming Problem with Convergence Rate O(1/k−−√)$O(1/\sqrt{k})$. *Soviet Mathematics Doklady*, 1983.

[2] I. Sutskever, J. Martens, G. Dahl, and G. Hinton. On the Importance of Initialization and Momentum in Deep Learning. *Proceedings of the 30th International Conference on Machine Learning*, 2013.

RMSprop

The **RMSprop** (`type: "RMSProp"`), suggested by Tieleman in a Coursera course lecture, is a gradient-based optimization method (like SGD). The update formulas are

MS((Wt)i)=δMS((Wt−1)i)+(1−δ)(∇L(Wt))2i(Wt+1)i=(Wt)i−α(∇L(Wt))iMS((Wt)i)−−−−−−−−−√

$$\mathrm{MS}((W_t)_i) = \delta\,\mathrm{MS}((W_{t-1})_i) + (1-\delta)(\nabla L(W_t))_i^2 \qquad (W_{t+1})_i = (W_t)_i - \alpha \frac{(\nabla L(W_t))_i}{\sqrt{\mathrm{MS}((W_t)_i)}}$$

The default value of δδ (`rms_decay`) is set to δ=0.99$\delta = 0.99$.

[1] T. Tieleman, and G. Hinton. RMSProp: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural Networks for Machine Learning.Technical report*, 2012.

Scaffolding

The solver scaffolding prepares the optimization method and initializes the model to be learned in `Solver::Presolve()` .

```
> caffe train -solver examples/mnist/lenet_solver.prototxt
I0902 13:35:56.474978 16020 caffe.cpp:90] Starting Optimization
```

```
I0902 13:35:56.475190 16020 solver.cpp:32] Initializing solver
from parameters:
test_iter: 100
test_interval: 500
base_lr: 0.01
display: 100
max_iter: 10000
lr_policy: "inv"
gamma: 0.0001
power: 0.75
momentum: 0.9
weight_decay: 0.0005
snapshot: 5000
snapshot_prefix: "examples/mnist/lenet"
solver_mode: GPU
net: "examples/mnist/lenet_train_test.prototxt"
```

Net initialization

```
 I0902 13:35:56.655681 16020 solver.cpp:72] Creating training
net from net file: examples/mnist/lenet_train_test.prototxt
[...]
I0902 13:35:56.656740 16020 net.cpp:56] Memory required for data:
0
I0902 13:35:56.656791 16020 net.cpp:67] Creating Layer mnist
I0902 13:35:56.656811 16020 net.cpp:356] mnist -> data
I0902 13:35:56.656846 16020 net.cpp:356] mnist -> label
I0902 13:35:56.656874 16020 net.cpp:96] Setting up mnist
I0902 13:35:56.694052 16020 data_layer.cpp:135] Opening lmdb
examples/mnist/mnist_train_lmdb
I0902 13:35:56.701062 16020 data_layer.cpp:195] output data size:
64,1,28,28
I0902 13:35:56.701146 16020 data_layer.cpp:236] Initializing
prefetch
I0902 13:35:56.701196 16020 data_layer.cpp:238] Prefetch
initialized.
I0902 13:35:56.701212 16020 net.cpp:103] Top shape: 64 1 28 28
(50176)
I0902 13:35:56.701230 16020 net.cpp:103] Top shape: 64 1 1 1 (64)
[...]
I0902 13:35:56.703737 16020 net.cpp:67] Creating Layer ip1
```

```
I0902 13:35:56.703753 16020 net.cpp:394] ip1 <- pool2
I0902 13:35:56.703778 16020 net.cpp:356] ip1 -> ip1
I0902 13:35:56.703797 16020 net.cpp:96] Setting up ip1
I0902 13:35:56.728127 16020 net.cpp:103] Top shape: 64 500 1 1
(32000)
I0902 13:35:56.728142 16020 net.cpp:113] Memory required for
data: 5039360
I0902 13:35:56.728175 16020 net.cpp:67] Creating Layer relu1
I0902 13:35:56.728194 16020 net.cpp:394] relu1 <- ip1
I0902 13:35:56.728219 16020 net.cpp:345] relu1 -> ip1 (in-place)
I0902 13:35:56.728240 16020 net.cpp:96] Setting up relu1
I0902 13:35:56.728256 16020 net.cpp:103] Top shape: 64 500 1 1
(32000)
I0902 13:35:56.728270 16020 net.cpp:113] Memory required for
data: 5167360
I0902 13:35:56.728287 16020 net.cpp:67] Creating Layer ip2
I0902 13:35:56.728304 16020 net.cpp:394] ip2 <- ip1
I0902 13:35:56.728333 16020 net.cpp:356] ip2 -> ip2
I0902 13:35:56.728356 16020 net.cpp:96] Setting up ip2
I0902 13:35:56.728690 16020 net.cpp:103] Top shape: 64 10 1 1
(640)
I0902 13:35:56.728705 16020 net.cpp:113] Memory required for
data: 5169920
I0902 13:35:56.728734 16020 net.cpp:67] Creating Layer loss
I0902 13:35:56.728747 16020 net.cpp:394] loss <- ip2
I0902 13:35:56.728767 16020 net.cpp:394] loss <- label
I0902 13:35:56.728786 16020 net.cpp:356] loss -> loss
I0902 13:35:56.728811 16020 net.cpp:96] Setting up loss
I0902 13:35:56.728837 16020 net.cpp:103] Top shape: 1 1 1 1 (1)
I0902 13:35:56.728849 16020 net.cpp:109]     with loss weight 1
I0902 13:35:56.728878 16020 net.cpp:113] Memory required for
data: 5169924
```

Loss

```
 I0902 13:35:56.728893 16020 net.cpp:170] loss needs backward
computation.
I0902 13:35:56.728909 16020 net.cpp:170] ip2 needs backward
computation.
I0902 13:35:56.728924 16020 net.cpp:170] relu1 needs backward
computation.
```

```
I0902 13:35:56.728938 16020 net.cpp:170] ip1 needs backward
computation.
I0902 13:35:56.728953 16020 net.cpp:170] pool2 needs backward
computation.
I0902 13:35:56.728970 16020 net.cpp:170] conv2 needs backward
computation.
I0902 13:35:56.728984 16020 net.cpp:170] pool1 needs backward
computation.
I0902 13:35:56.728998 16020 net.cpp:170] conv1 needs backward
computation.
I0902 13:35:56.729014 16020 net.cpp:172] mnist does not need
backward computation.
I0902 13:35:56.729027 16020 net.cpp:208] This network produces
output loss
I0902 13:35:56.729053 16020 net.cpp:467] Collecting Learning Rate
and Weight Decay.
I0902 13:35:56.729071 16020 net.cpp:219] Network initialization
done.
I0902 13:35:56.729085 16020 net.cpp:220] Memory required for
data: 5169924
I0902 13:35:56.729277 16020 solver.cpp:156] Creating test net
(#0) specified by net file: examples/mnist
/lenet_train_test.prototxt
```

Completion

```
 I0902 13:35:56.806970 16020 solver.cpp:46] Solver scaffolding
done.
I0902 13:35:56.806984 16020 solver.cpp:165] Solving LeNet
```

Updating Parameters

The actual weight update is made by the solver then applied to the net parameters in `Solver::ComputeUpdateValue()`. The `ComputeUpdateValue` method incorporates any weight decay r(W)$r(W)$ into the weight gradients (which currently just contain the error gradients) to get the final gradient with respect to each network weight. Then these gradients are scaled by the learning rate α and the update to subtract is stored in each parameter Blob's `diff` field. Finally, the `Blob::Update` method is called on each parameter blob, which performs the final update (subtracting the Blob's `diff` from its `data`).

Snapshotting and Resuming

The solver snapshots the weights and its own state during training in `Solver::Snapshot()` and `Solver::SnapshotSolverState()`. The weight snapshots export the learned model while the solver snapshots allow training to be resumed from a given point. Training is resumed by `Solver::Restore()` and `Solver::RestoreSolverState()`.

Weights are saved without extension while solver states are saved with `.solverstate` extension. Both files will have an `_iter_N` suffix for the snapshot iteration number.

Snapshotting is configured by:

```
 # The snapshot interval in iterations.
snapshot: 5000
# File path prefix for snapshotting model weights and solver
state.
# Note: this is relative to the invocation of the `caffe`
utility, not the
# solver definition file.
snapshot_prefix: "/path/to/model"
# Snapshot the diff along with the weights. This can help
debugging training
# but takes more storage.
snapshot_diff: false
# A final snapshot is saved at the end of training unless
# this flag is set to false. The default is true.
snapshot_after_train: true
```

in the solver definition prototxt.