TTIC 31230, Fundamentals of Deep Learning

David McAllester, Winter 2020

Stochastic Gradient Descent (SGD)

The Classical Convergence Theorem

Momentum, RMSProp, and Adam

Decoupling SGD Hyperparameters

Vanilla SGD

$$\Phi \texttt{ -= } \eta \hat{g}$$

$$\hat{g} = E_{(x,y)\sim \text{Batch}} \nabla_\Phi \text{loss}(\Phi, x, y)$$

$$g = E_{(x,y)\sim \text{Pop}} \nabla_\Phi \text{loss}(\Phi, x, y)$$

Issues

- **Gradient Estimation.** The accuracy of $\hat{g}$ as an estimate of g.
- **Gradient Drift (second order structure).** The fact that g changes as the parameters change.
- **Convergence.** To converge to a local optimum the learning rate must be gradually reduced toward zero.
- **Exploration.** Since deep models are non-convex we need to search over the parameter space. SGD can behave like MCMC.

A One Dimensional Example

Suppose that y is a scalar, and consider

$$\begin{aligned}
\text{loss}(\beta, y) &= \frac{1}{2}(\beta - y)^2 \\
g &= \nabla_\beta \, E_{y \sim \text{Pop}} \, \frac{1}{2}(\beta - y)^2 \\
&= \beta - E_{y \sim \text{Pop}} \, y \\
\hat{g} &= \beta - E_{y \sim \text{Batch}} \, y
\end{aligned}$$

Even if β is optimal, for a finite batch we will have $\hat{g} \neq 0$.

The Classical Convergence Theorem

$$\Phi \texttt{ -= } \eta_t \nabla_\Phi \operatorname{loss}(\Phi, x_t, y_t)$$

For "sufficiently smooth" non-negative loss with

$$\eta_t \geq 0 \qquad \lim_{t \to \infty} \eta_t = 0 \qquad \sum_t \eta_t = \infty \qquad \sum_t \eta_t^2 < \infty$$

we have that the training loss $E_{(x,y)\sim\text{Train}} \operatorname{loss}(\Phi, x, t)$ converges to a limit and any limit point of the sequence Φ_t is a stationary point in the sense that $\nabla_\Phi E_{(x,y)\sim\text{Train}} \operatorname{loss}(\Phi, x, t) = 0$.

Rigor Police: One can construct cases where Φ diverges to infinity, converges to a saddle point, or even converges to a limit cycle.

Physicist's Proof of the Convergence Theorem

Since $\lim_{t \to 0} \eta_t = 0$ we will eventually get to arbitrarilly small learning rates.

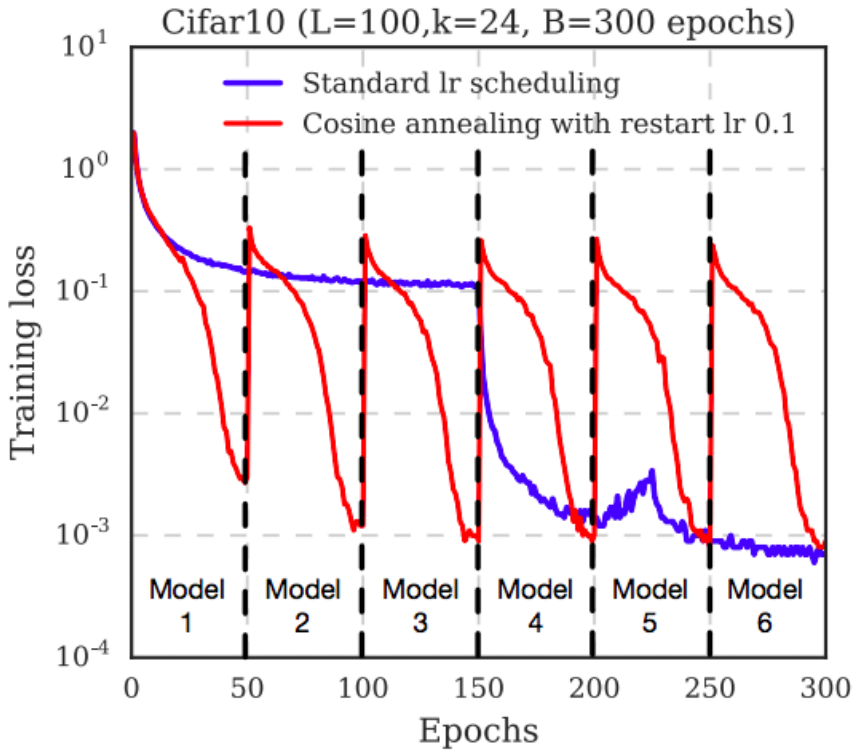
For sufficiently small learning rates any meaningful update of the parameters will be based on an arbitrarily large sample of gradients at essentially the same parameter value.

An arbitrarily large sample will become arbitrarily accurate as an estimate of the full gradient.

But since $\sum_t \eta_t = \infty$, no matter how small the learning rate gets, we still can make arbitrarily large motions in parameter space.

SGD as a form of MCMC

Learning Rate as a Temperature Parameter



Gao Huang et. al., ICLR 2017

Decoupling the Learning Rate η from the Batch Size B

Vanilla SGD:

$$\Phi_{t+1} \texttt{ -= } \eta \hat{g}_t$$

$$\hat{g}_t = \frac{1}{B} \sum_b \hat{g}_{t,b}$$

Where $\hat{g}_{t,b}$ is the gradient of the element b of the batch.

Here the effect of $\hat{b}_{t,b}$ on Φ is $\frac{\eta}{B} \hat{g}_{t,b}$.

To make the effect of $\hat{g}_{t,b}$ independent of B we can use

$$\eta = B\eta_0$$

where η_0 is optimal for vanilla SGD with $B = 1$.

Decoupling η from B

Recent work has show that using $\eta = B\eta_0$ leads to effective learning with very large (highly parallel) batches.

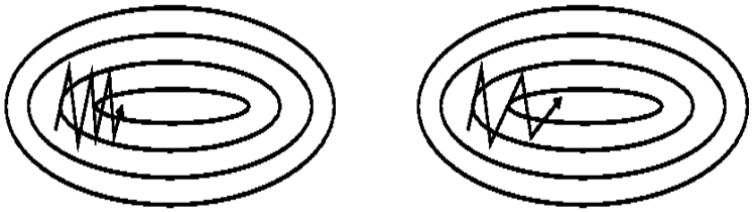
Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour, Goyal et al., 2017.

Momentum

$$v_t = \mu v_{t-1} + \eta * \hat{g}_t \quad \mu \text{ is typically .9 or .99}$$

$$\Phi_{t+1} = \Phi_t - v_t$$

The theory of momentum is generally given in terms of second order structure and total gradients (GD rather than SGD).



Rudin's blog

Momentum

GD analyses seem unlikely to apply to SGD.

Second order analyses are of questionable value for SDG in very large dimension.

But momentum is widely used in SGD.

To better understand momentum we will rewrite it as a running average.

Running Averages

Consider a sequence $x_1, x_2, x_3, \ldots$.

For $t \geq N$, consider the average of the N most recent values.

$$\overline{x}_t = \frac{1}{N} \sum_{\tilde{t}=t-N+1}^{t} x_{\tilde{t}}$$

This can be approximated efficiently with

$$\tilde{x}_0 = 0$$

$$\tilde{x}_t = \left(1 - \frac{1}{N}\right) \tilde{x}_{t-1} + \left(\frac{1}{N}\right) x_t$$

Deep Learning Convention for Running Averages

In deep learning a running average

$$\tilde{x}_t = \left(1 - \frac{1}{N}\right)\tilde{x}_{t-1} + \left(\frac{1}{N}\right)x_t$$

is written as

$$\tilde{x}_t = \beta\tilde{x}_{t-1} + (1-\beta)x_t$$

where

$$\beta = 1 - 1/N$$

Typical values for β are .9, .99 or .999 corresponding to N being 10, 100 or 1000.

It will be convenient here to use N rather than β.

Momentum as a Running Average

$$v_t = \mu v_{t-1} + \eta \hat{g}_t$$

$$= \left(1 - \frac{1}{N_g}\right) v_{t-1} + \frac{1}{N_g}(N_g \eta \hat{g}_t)$$

Here we rewrite this in terms of a direct running average of the gradient.

$$\tilde{g}_t = \left(1 - \frac{1}{N_g}\right) \tilde{g}_{t-1} + \frac{1}{N_g}\hat{g}_t$$

Since averaging is a linear operation, we get

$$v_t = N_g \eta \tilde{g}_t$$

Momentum as a Running Average

We have now shown that the momentum update is

$$\Phi_{t+1} = \Phi_t - v_t = \Phi_t - N_g \eta \tilde{g}_t$$

Hence momentum can be written as

$$\tilde{g}_t = \left(1 - \frac{1}{N_g}\right) \tilde{g}_{t-1} + \frac{1}{N_g} \hat{g}_t$$

$$\Phi_{t+1} = \Phi_t - \tilde{\eta} \tilde{g}_t$$

$$\tilde{\eta} = N_g \eta$$

Generalizing $\eta = B\eta_0$

We will use the general principle that the effect of a single batch element $\hat{g}_{t,b}$ on the model should be $\eta_0 \hat{g}_{t,b}$ where η_0 is the optimal learning rate for Vanilla SGD with batch size 1.

This is consisting with the empirically validated scaling of η with batch size.

It is also intuitively motivated by the idea that the model is changing slowly and delaying the effect of $\hat{g}_{t,b}$ does not change the total effect it should have.

Decoupling η from N_g

$$\tilde{g}_t = \left(1 - \frac{1}{N_g}\right) \tilde{g}_{t-1} + \frac{1}{N_g} \hat{g}_t$$

$$\Phi_{t+1} = \Phi_t - \tilde{\eta} \tilde{g}_t$$

For $\Delta t >> N_g$, the effect of $\hat{g}_t$ on $\Phi_{t+\Delta t}$ is approximately

$$\frac{\tilde{\eta} \hat{g}_t}{N_g} \sum_{i=0}^{\infty} \left(1 - \frac{1}{N_g}\right)^i = \frac{\tilde{\eta} \hat{g}_t}{N_g} N_g = \tilde{\eta} \hat{g}_t$$

So the effect of $\hat{g}_t$ on the model is $\tilde{\eta} \hat{g}_t$ independent of N_g.

This suggests that the optimal value of $\tilde{\eta}$ is independent of N_g.

Decoupling η from N_g

Using the same value of $\tilde{\eta}$ independent of N_g gives

$$\tilde{\eta} = B\eta_0 = N_g\eta$$

where, as before, η_0 is the optimal learning rate for Vanilla SGD with $B = 1$.

Solving for η we get

$$\eta = \frac{B\eta_0}{N_g}$$

Don't Decay the Learning Rate, Increase the Batch Size, Smith et al., 2018

Decoupling η from N_g

$$\eta = \frac{B\eta_0}{N_g}$$

One should set β to be large enough so as to saturate the hardware (your GPUs should be running at maximum FLOPs).

The optimnal value of η_0 should be fairly insensitive to the choice of B and N_g.

Decoupling N_g from B

Let N_g^0 be a momentum parameter appropriate for $B = 1$.

Momentum is based on a moving average of gradients

$$\tilde{g}_{t+1} = \left(1 - \frac{1}{N_g}\right) \tilde{g}_t + \frac{1}{N_g} \hat{g}_t$$

In the presence of minibatching this is (approximately) averaging over N_g minibatches.

But to average over N_g^0 individual gradients we should use

$$N_g = \frac{N_g^0}{B}$$

Putting It Together

$$N_g = \min\left(1, N_g^0/B\right)$$

$$\eta = \frac{B\eta_0}{N_g}$$

We can take B to be as large as memory allows and then tune N_g^0 and η_0 rather than N_g and η.

The optimal values for N_g^0 and η_0 should be relatively independent of the batch size B.

RMSProp and Adam

RMSProp and Adam are "adaptive" SGD methods — the effective learning rate is computed from statistics of the data rather than set as a fixed hyper-parameter (although the adaptation algorithm itself still has hyper-parameters).

Different effective learning rates are used for different model parameters.

Adam is typically used in NLP while Vanilla SGD is typically used in vision. This may be related to the fact that batch normalization is used in vision but not in NLP.

RMSProp

RMSProp is based on a running average of $\hat{g}[i]^2$ for each scalar model parameter i.

$$s_t[i] = \left(1 - \frac{1}{N_s}\right) s_{t-1}[i] + \frac{1}{N_s}\hat{g}_t[i]^2 \quad N_s \text{ typically 100 or 1000}$$

$$\Phi_{t+1}[i] = \Phi_t[i] - \frac{\eta}{\sqrt{s_t[i]} + \epsilon}\ \hat{g}_t[i]$$

RMSProp

The second moment of a scalar random variable x is $E\ x^2$

The variance σ^2 of x is $E\ (x - \mu)^2$ with $\mu = E\ x$.

RMSProp uses an estimate $s[i]$ of the second moment of the random scalar $\hat{g}[i]$.

For $g[i]$ small $s[i]$ approximates the variance of $\hat{g}[i]$.

There is a "centering" option in PyTorch RMSProp that switches from the second moment to the variance.

RMSProp Motivation

$$\Phi_{t+1}[i] = \Phi_t[i] - \frac{\eta}{\sqrt{s_t[i]} + \epsilon} \hat{g}_t[i]$$

One interpretation of RMSProp is that a low variance gradient has less statistical uncertainty and hence needs less averaging before making the update.

RMSProp is Theoretically Mysterious

$$\Phi[i] \texttt{ -= } \eta \frac{\hat{g}[i]}{\sigma[i]} \quad (1) \qquad\qquad \Phi[i] \texttt{ -= } \eta \frac{\hat{g}[i]}{\sigma^2[i]} \quad (2)$$

Although (1) seems to work better, (2) is better motivated theoretically. To see this we can consider units.

If parameters have units of "weight", and loss is in bits, then (2) type checks with η having units of inverse bits — the numerical value of η has no dependence on the choice of the weight unit.

Consistent with the dimensional analysis, many theoretical analyses support (2) over (1) contrary to apparent empirical performance.

Adam — Adaptive Momentum

Adam combines momentum and RMSProp (although PyTorch RMSProp also supports momentum).

Adam also uses "bias correction" which probably accounts for it's popularity over RMSProp in practice.

Bias Correction

Consider a standard moving average.

$$\tilde{x}_0 = 0$$

$$\tilde{x}_t = \left(1 - \frac{1}{N}\right) \tilde{x}_{t-1} + \left(\frac{1}{N}\right) x_t$$

For $t < N$ the average $\tilde{x}_t$ will be strongly biased toward zero.

Bias Correction

The following running average maintains the invariant that $\tilde{x}_t$ is exactly the average of $x_1, \ldots, x_t$.

$$\tilde{x}_t = \left(\frac{t-1}{t}\right)\tilde{x}_{t-1} + \left(\frac{1}{t}\right)x_t$$

$$= \left(1 - \frac{1}{t}\right)\tilde{x}_{t-1} + \left(\frac{1}{t}\right)x_t$$

We now have $\tilde{x}_1 = x_1$ independent of any x_0.

But this fails to track a moving average for $t >> N$.

Bias Correction

The following avoids the initial bias toward zero while still tracking a moving average.

$$\tilde{x}_t = \left(1 - \frac{1}{\min(N,t)}\right)\tilde{x}_{t-1} + \left(\frac{1}{\min(N,t)}\right)x_t$$

The published version of Adam has a more obscure form of bias correction which yields essentially the same effect.

Adam (simplified)

$$\tilde{g}_t[i] = \left(1 - \frac{1}{\min(t, N_g)}\right) \tilde{g}_{t-1}[i] + \frac{1}{\min(t, N_g)} \hat{g}_t[i]$$

$$s_t[i] = \left(1 - \frac{1}{\min(t, N_s)}\right) s_{t-1}[i] + \frac{1}{\min(t, N_s)} \hat{g}_t[i]^2$$

$$\Phi_{t+1}[i] = \Phi_t - \frac{\eta}{\sqrt{s_t[i]} + \epsilon} \; \tilde{g}_t[i]$$

Decoupling Hyperparametera

The following reparameterization should be helpful for Adam.

$$N_g = min(1, N_g^0/B)$$

$$\eta = \epsilon B \eta_0$$

Empirically, tuning ϵ is important.

Some coupling remains between η_0 and ϵ.

N_s should also be adapted to B but this is problematic — see the next slide.

Making $s_t[i]$ batch size invariant

rather than

$$s_t[i] = \left(1 - \frac{1}{N_s}\right) s_{t-1}[i] + \frac{1}{N_s} \hat{g}_t[i]^2$$

we would like

$$s_t[i] = \left(1 - \frac{1}{N_s}\right) s_{t-1}[i] + \frac{1}{N_s} \left(\frac{1}{B} \sum_b \hat{g}_{t,b}[i]^2\right)$$

$$N_s = \min\left(1, N_s^0/B\right) \quad N_s^0 \text{ optimal for } B = 1$$

Making $s_t[i]$ Batch Size Invariant

In PyTorch this is difficult because "optimizers" are defined as a function of $\hat{g}_t$.

$\hat{g}_t[i]$ is not sufficient for computing $\sum_b \ \hat{g}_{t,b}[i]^2$.

To compute $\sum_b \ \hat{g}_{t,b}[i]^2$ we need to modify the backward method of all PyTorch objects!

Summary

We have considered Vanilla SGD, Momentum, RMSProp and Adam.

Vanilla tends to be used in vision while Adam tends to be used in NLP.

Reparameterization of the PyTorch hyperparameters can decouple hyperparameters simplifying hyperparameter search.

END