50.021 – AI

Alex

Week 04: Smart Optimizers – Better (?) ways to apply gradients

[The following notes are compiled from various sources such as textbooks, lecture materials, Web resources and are shared for academic purposes only, intended for use by students registered for a specific course. In the interest of brevity, every source is not cited. The compiler of these notes gratefully acknowledges all such sources.]

Key takeaways:

Be able to explain the main ideas behind:

- weight decay and its equivalence to ℓ_2-regularization
- momentum term
- RMSprop
- Adam
- updates in momentum term, RMSprop , Adam use exponential moving averages (EMA)
- AdamW

1 Better gradients

good source: Sebastian Ruder, An overview of gradient descent optimization algorithms `https://arxiv.org/abs/1609.04747` `http://sebastianruder.com/optimizing-gradient-descent/index.html`

What we had for optimization: want to find a parameter w corresponding to a mapping $f_w : x \mapsto f(x) \in \mathcal{Y}$

$$\hat{E}(w, L, 1, n) = \frac{1}{n}\sum_{i=1}^{n} L(f_w(x_i), y_i)$$
$$\text{argmin}_w \hat{E}(f_w, L, 1, n)$$

Here we made in the loss the dependency on the sample set $1, \ldots, n$ explicit.
Basic Algorithm idea **(Gradient Descent)**:

- initialize start vector w_0 as something, step size parameter η
- run while loop until vector changes very little, do at iteration t:
  - $w_{t+1} = w_t - \eta \nabla_w \hat{E}(w_t, L, 1, n) = w_t - \text{learningrate} \cdot \frac{dE}{dw}(w_t)$
  - compute change to last: $\|w_{t+1} - w_t\|$

Problem is: deep neural networks have many parameters - w has hundred thousands of dimensions. Need more tricks to get it all working well.

1.1 How to choose a learning rate

First question: How to choose the learning rate?

Answer: there is no general solution for it - try and error on your problem.

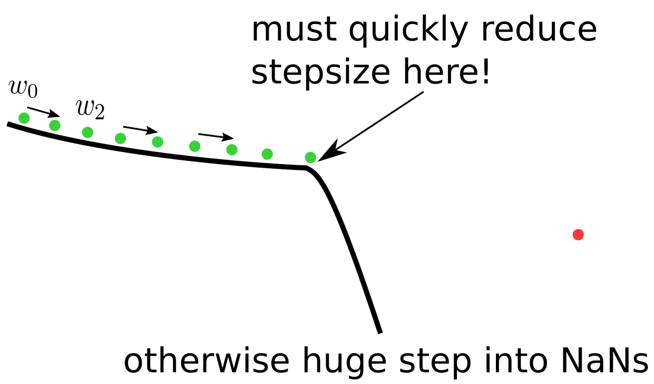
Problems with fixed learning rate: `quadform.py`

- DIVERGENCE if learning rate too high - (see example in past lecture)
- in a flat region steps can be very small:

  Observation: size of update of weights, as measured by euclidean length is proportional to the norm of the gradient:

$$w_{t+1} = w_t - \eta_t \nabla_w \hat{E}(w_t, L, 1, n)$$
$$\|w_{t+1} - w_t\| = \eta_t \|\nabla_w \hat{E}(w_t, L, 1, n)\|$$

  So in a flat region with $\|\nabla_w \hat{E}(w_t, L, 1, n)\| \approx 0$, the steps taken are very small.
- long flat region followed by a steep decline, want to go fast first, but must go slow in the steep part - a constant stepsize is either too slow at the start, or too fast at the end



- typical solution **step-wise learning rate decay**: not constant learning rate η but reduce learning rate by multiplying with a constant $\gamma \in (0,1)$ once every K steps:

$$\eta_{t+1} = \begin{cases} \eta_t \cdot \gamma, \ 0 < \gamma < 1 & \text{if } t = c \cdot K \text{ for some } c = 1,2,3,\ldots \\ \eta_t & \text{else} \end{cases}$$

other solution – **polynomial learning rate decay**, (but in deep learning often too fast decrease of η_t)

$$\eta_t = \frac{\eta_0}{t^\alpha}, \ \alpha > 0$$

Important:

Learning rate reduction schemes to know:

- stepwise learning rate reduction
- polynomial learning rate reduction
- SGD with warm restarts `https://arxiv.org/pdf/1608.03983.pdf` Loshchilov and Hutter ICLR 2017

  `https://towardsdatascience.com/https-medium-com-reina-wang-tw-stochastic-gradient-descent-with-restarts-5f511975`

  - for the i-th run decrease learningrate from $\eta^{(i)}_{max}$ to $\eta^{(i)}_{min}$ by a cosine quarter wave:

  $$\eta_t = \eta^{(i)}_{min} + (\eta^{(i)}_{max} - \eta^{(i)}_{min})\frac{1}{2}(1 + \cos(\pi\frac{T_{cur}}{T_i}))$$

  - after that increase the number of epochs by a factor $T_{i+1} = T_i \cdot T_{mult}$ that are needed to get from $\eta^{(i)}_{max}$ to $\eta^{(i)}_{min}$

a paper that makes use of that for generating an ensemble along the restarts: `https://arxiv.org/pdf/1704.00109.pdf`

1.2 Weight decay

Replace

$$\begin{aligned} w_{t+1} &= w_t & &-\eta_t \nabla_w \hat{E}(w_t, L) \text{ by} \\ w_{t+1} &= w_t(1 - \lambda\eta_t) & &-\eta_t \nabla_w \hat{E}(w_t, L) \end{aligned}$$

shrinks weight towards zero. Comes from quadratic regularization:

$$\begin{aligned}
\hat{E}_{Reg}(w, L) &= \frac{1}{n}\sum_{i=1}^{n} L(f_w(x_i), y_i) && + \frac{1}{2}\lambda\eta_t\|w\|^2 \\
\nabla_w \hat{E}_{Reg}(w, L) &= \nabla_w \frac{1}{n}\sum_{i=1}^{n} L(f_w(x_i), y_i) && + \nabla_w \frac{1}{2}\lambda\eta_t\|w\|^2 \\
\nabla_w \hat{E}_{Reg}(w, L) &= \nabla_w \hat{E}(w, L) && + \lambda\eta_t w \\
\text{therefore: } w_{t+1} &= w_t - \eta_t \nabla_w \hat{E}(w, L) - \lambda\eta_t w \\
\text{therefore: } w_{t+1} &= w_t(1 - \lambda\eta_t) - \eta_t \nabla_w \hat{E}(w, L)
\end{aligned}$$

Important:

for SGD weight decay is the same as ℓ_2-regularization.
Weight decay in general is the multiplication of a weight with a small number $w = w \cdot \gamma, \gamma \in (0, 1)$

1.3 Momentum term

many more heuristics replace $w_{t+1} = w_t - \eta_t \nabla_w \hat{E}(w_t, L)$ by something related to it.

$$\begin{aligned}
m_0 &= 0, \alpha \in (0, 1) \\
m_{t+1} &= \alpha m_t + \eta_t \nabla_w \hat{E}(w_t, L) \\
w_{t+1} &= w_t - m_{t+1}
\end{aligned}$$

- how: compute an average m_{t+1} between current gradient $\eta_t \nabla_w \hat{E}(w_t, L)$ and *gradients from the past* m_t. use this average for updating weights
- acts as a memory for gradients in the past, applied gradient is stabilized by an average from the past
- it can help in flat valleys because it remember the past bigger stepsize from past steps
- reduce influence of too big gradients when taking an unlucky step into a steeply mountainous region resulting in high gradients – gradient still stays small
- with one more parameter α

What does the momentum compute? Assume $\eta_t = \eta$ is constant.

Lets shorten: $g_t = \nabla_w \hat{E}(w_t, L)$

$$\begin{aligned}
m_1 &= \alpha m_0 + \eta g_0 = \eta g_0 \\
m_2 &= \alpha m_1 + \eta g_1 = \alpha^1 \eta g_0 + \eta g_1 \\
m_3 &= \alpha m_2 + \eta g_2 = \alpha^2 \eta g_0 + \alpha^1 \eta g_1 + \eta g_2 \\
m_4 &= \alpha m_3 + \eta g_3 = \alpha^3 \eta g_0 + \alpha^2 \eta g_1 + \alpha^1 \eta g_2 + \eta g_3 \\
m_5 &= \alpha m_4 + \eta g_4 = \alpha^4 \eta g_0 + \alpha^3 \eta g_1 + \alpha^2 \eta g_2 + \alpha^1 \eta g_3 + \eta g_4
\end{aligned}$$

general rule:

$$m_t = \eta \left(\sum_{s=0}^{t-1} \alpha^{t-1-s} g_s \right)$$

What does this represent: consider $g_0, g_1, g_2, \ldots$ as a time series. Then

- m_t is a weighted average up to multiplication with a constant.
- the weights of this average decrease exponential as we go back into the past

Vanilla average over $g_0, g_1, g_2, \ldots$:

$$\frac{1}{t} \sum_{s=0}^{t-1} g_s = \sum_{s=0}^{t-1} \frac{1}{t} g_s$$

a weighted average would be:

$$\sum_{s=0}^{t-1} w_s g_s$$

$$w_s \geq 0, \sum_{s=0}^{t-1} w_s = 1$$

Vanilla average is weighted with constant (time-independent weights): $w_s = \frac{1}{t}$. This satisfies $\sum_{s=0}^{t-1} w_s = \sum_{s=0}^{t-1} \frac{1}{t} = 1$

For the momentum term:

$$\begin{aligned}
&\alpha^{t-1-s} \geq 0 \\
&\sum_{s=0}^{t-1} \alpha^{t-1-s} = \alpha^{t-1} + \alpha^{t-2} + \alpha^{t-3} + \ldots + \alpha^2 + \alpha^1 + \alpha^0 \\
&\qquad\qquad = \sum_{s=0}^{t-1} \alpha^s = \frac{1-\alpha^t}{1-\alpha}
\end{aligned}$$

So it –almost– sums up to one. It is a weighted average up to division of weights by $\frac{1-\alpha^t}{1-\alpha}$.

Exponential decay from terms in the past: Earliest term:

$$s = 0 \Rightarrow \alpha^{t-1-s} = \alpha^{t-1}$$

Since $0 < \alpha < 1$ this is a very small term. Latest term has weight 1.
In summary: it is an average - so it can dampen against single bad gradients,

and weights for gradients decrease exponentially towards the past. So it looks more at the recent past. In practice often $\alpha = 0.9$ – so the past has stronger weight than the present.

Important:

SGD with momentum and $\alpha = 0.9$ and weight decay is a very common baseline choice

$$m_{t+1} = \alpha m_t + \eta \nabla_w \hat{E}(w_t, L)$$
$$w_{t+1} = w_t - m_{t+1} - \eta \beta w_t$$

1.4 Exponential moving average (EMA)

For a time series g_s the term

$$\begin{aligned} \mathrm{EMA}(g_s)_0 &= 0 && +(1-\alpha)g_0 \\ \mathrm{EMA}(g_s)_t &= \alpha \mathrm{EMA}(g_s)_{t-1} && +(1-\alpha)g_t \end{aligned}$$

defines an exponential moving average. Moving – because weights are high for recent past.
The recursion yields here

$$\begin{aligned} \mathrm{EMA}(g_s)_0 &= \alpha^0(1-\alpha)g_0 \\ \mathrm{EMA}(g_s)_1 &= \alpha^1(1-\alpha)g_0 + (1-\alpha)g_1 \\ \mathrm{EMA}(g_s)_2 &= \alpha^2(1-\alpha)g_0 + \alpha^1(1-\alpha)g_1 + (1-\alpha)g_2 \\ \mathrm{EMA}(g_s)_3 &= \alpha^3(1-\alpha)g_0 + \alpha^2(1-\alpha)g_1 + \alpha^1(1-\alpha)g_2 + (1-\alpha)g_3 \\ \mathrm{EMA}(g_s)_t &= \sum_{s=0}^{t} \alpha^{t-s}(1-\alpha)g_s \end{aligned}$$

The weights of $\mathrm{EMA}(g_s)_t$ sum up to $1 - \alpha^{t+1}$.

1.5 RMSProp

An idea to deal with the flat regions – Unpublished method by Geoffrey Hinton.

Observation: size of update of weights, as measured by euclidean length is proportional to the norm of the gradient:

$$g_t = \nabla_w \hat{E}(w_t, L)$$
$$w_{t+1} = w_t - \eta_t g_t$$
$$\|w_{t+1} - w_t\| = \eta_t \|g_t\|$$

So in a flat region with $\|g_t\| \approx 0$, the steps taken are very small.

First idea: use gradient divided by norm of gradient

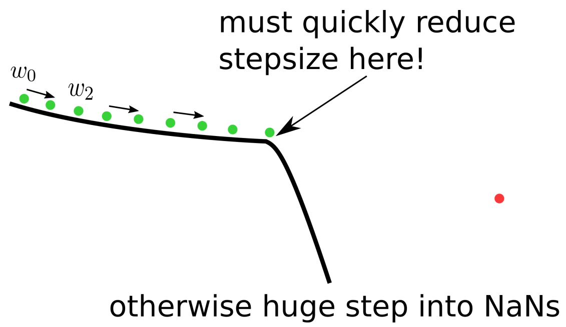
$$w_{t+1} = w_t - \eta_t \frac{g_t}{\|g_t\|}$$

Problem with this: whether one is in a looong flat region or not cannot be decided by looking at a single gradient at the current point - need to look a bit more intor the past.

So use an average of norms of gradients from the past, and divide by them. Divide by $\mathrm{EMA}(\cdot)_t$ of norms of gradients:

$$w_{t+1} = w_t - \eta_t \frac{g_t}{\mathrm{EMA}(\|g_s\|)_t}$$

Idea: flat valley, for many time steps s around the current time step t norms of gradients are small, so EMA will be small. Dividing by a small term makes the stepsize bigger.
Still not perfect: We need to reduce the stepsize fast when we enter more steep regions. That means: if a current gradient norm $\|g_t\|$ at time t is large, the EMA needs to become large quickly (so that dividing by a large EMA leads to a small step)!



Squared norms are better, as squares are more sensitive to large outliers in a sum (x^2 grows quicker than x). so use $\|g_t\|^2$ - squared norms in the EMA (and take a root of the EMA).

$$w_{t+1} = w_t - \eta_t \frac{g_t}{\sqrt{\mathrm{EMA}(\|g_s\|^2)_t}}$$

Still not perfect. what is if all gradients are near-zero? Huge step into the numerical void. Better: add a small ϵ

$$w_{t+1} = w_t - \eta_t \frac{g_t}{\sqrt{\mathrm{EMA}(\|g_s\|^2)_t + \epsilon}}$$

Now upscaling factor is limited by $\frac{1}{\sqrt{\epsilon}}$

This **RMSProp Algorithm** can be rewritten in an iterative form, which is easier to code:

Parameters: α, ϵ, η

$$
\begin{aligned}
d_0 &= 0 \\
\text{compute } g_t &:= \nabla_w \hat{E}(w_t, L) \\
d_t &= \alpha d_{t-1} + (1-\alpha)\|g_t\|^2 \qquad \# \; d_t \; is \; \mathrm{EMA}(\|g_s\|^2)_t \\
w_{t+1} &= w_t - \eta_t \frac{g_t}{\sqrt{d_t + \epsilon}}
\end{aligned}
$$

can be remembered as:

maintain an EMA for squared norms of gradient, divide gradient by the square-root of it plus some stabilizing ϵ. Use this for update of weights.

Its effect can be understood as:

- divide gradient dE/dw by a history of gradient norms with time-limited horizon
- upscales stepsize in flat region
- downscales stepsize when it becomes mountainous

side note

A mild weight decay with RMSprop (around 1e-6 for a learning rate of 1e-3) proved useful for two reinforcement learning tasks from the openAI gym. Adam (below) worked also.

1.6 Adam

Very popular. A Must know.

Similar to a combination RMSprop with Momentum Term but Two ideas as improvement over RMSprop.

How would RMSprop with Momentum Term look like in step t?

compute $g_t := \nabla_w \hat{E}(w_t, L)$

$$d_t = \alpha_1 d_{t-1} + (1-\alpha_1)\|g_t\|^2 \quad \# \; d_t \; is \; EMA(\|g_s\|^2)_t$$

$$rpropterm = \frac{g_t}{\sqrt{d_t} + \epsilon}$$

In RMSProp one would apply *rpropterm* to update the weights w_t with a stepsize η_t. Now one replaces in *rpropterm* the gradient g_t by its momentum m_t:

$$m_t = \alpha_2 m_{t-1} + (1-\alpha_2) g_t$$
$$w_{t+1} = w_t - \eta_t \frac{m_t}{\sqrt{d_t} + \epsilon}$$

The two improvements are made in Adam over the algorithm above:

1. normalize every dimension of the update separately – dont use the norm of the gradient, but the square of every single dimension
2. turn all used/defined terms which use an EMA into a true weighted average by multiplying them with the appropriate constant $\frac{1}{1-\alpha^t}$

We explain both steps in detail.

Point 1. normalize every dimension of the update separately:

The gradient g_t is a vector $g_t = (g_t^{(1)}, \ldots, g_t^{(d)}, \ldots, g_t^{(D)})$. When computing *rpropterm* above every dimension d of g_t is scaled by the same constant:

$$\frac{1}{\sqrt{EMA(\|g_s\|^2)_t} + \epsilon} = \frac{1}{\sqrt{d_t} + \epsilon}$$

In Adam one computes an EMA for every dimension $g_t[d]$ of the gradient. One uses the square $(g_t[d])^2$ of the gradient in analogy to the squared norm $\|g_t\|^2$.

$$d_t = \alpha_1 d_{t-1} + (1-\alpha_1)(g_t[d])^2$$

This d_t is now a vector!

Using only 1. the algorithm would look like that:

$$
\begin{aligned}
\text{compute } g_t &:= \nabla_w \hat{E}(w_t, L) \\
d_t &= \alpha_1 d_{t-1} + (1-\alpha_1)(g_t[d])^2 \\
m_t &= \alpha_2 m_{t-1} + (1-\alpha_2) g_t \\
w_{t+1} &= w_t - \eta_t \frac{m_t}{\sqrt{d_t} + \epsilon} \\
1/\sqrt{d_t} &: \text{ (element-wise division for every dimension } d\text{)}
\end{aligned}
$$

Point 2. turn all used/defined terms which use an EMA into a true weighted average by multiplying them with an appropriate constant:

This is based on the observation, that the weights of every $EMA(u_s)_t$ sum up to $1 - \alpha^{t+1}$.

Therefore whenever applying an EMA term, it must be divided by $1 - \alpha^{t+1}$, in order to yield a true weighted average. An EMA is used here in two steps: once when computing $term$, a second time when computing w_{t+1}.

The final **ADAM algorithm** is:

Parameters $\eta, \epsilon, \alpha_1, \alpha_2$

$$
\begin{aligned}
m_0 &= 0, d_0 = 0 \\
\text{compute } g_t &:= \nabla_w \hat{E}(w_t, L) \\
d_t &= \alpha_1 d_{t-1} + (1-\alpha_1)(g_t[d])^2 \quad \# element-wise \\
m_t &= \alpha_2 m_{t-1} + (1-\alpha_2) g_t \\
c_{t,1} &= 1 - \alpha_1^t, c_{t,2} = 1 - \alpha_2^t \\
w_{t+1} &= w_t - \eta_t \frac{m_t / c_{t,1}}{\sqrt{d_t / c_{t,2}} + \epsilon} \\
1/\sqrt{d_t} &: \text{ (element-wise division for every dimension } d\text{)}
\end{aligned}
$$

can be remembered as:

- a combination of RMSProp with momentum
- one momentum term for the gradient (one EMA),
- and another EMA for the element-wise squared gradient
- plus dividing both EMAs by those constants that makes them true weighted averages.
- Use for weight update: the (constant-adjusted) momentum divided element-wise by the (constant-adjusted) square-root of the EMA for the element-wise squared gradient $+\epsilon$.

1.7 AdamW: Adam with decoupled weight decay

`https://arxiv.org/abs/1711.05101` Loshchilov & Hutter, ICLR 2019

Parameters $\eta, \epsilon, \alpha_1, \alpha_2$

$$
\begin{aligned}
m_0 &= 0, d_0 = 0 \\
\text{compute } g_t &:= \nabla_w \hat{E}(w_t, L) \\
d_t &= \alpha_1 d_{t-1} + (1-\alpha_1)(g_t[d])^2 \quad \#element-wise \\
m_t &= \alpha_2 m_{t-1} + (1-\alpha_2) g_t \\
c_{t,1} &= 1-\alpha_1^t, c_{t,2} = 1-\alpha_2^t \\
w_{t+1} &= w_t - \eta_t \frac{m_t/c_{t,1}}{\sqrt{d_t/c_{t,2}}+\epsilon} - \lambda \eta_t w_t \\
&\quad 1/\sqrt{d_t}: \ \text{(element-wise division for every dimension } d\text{)}
\end{aligned}
$$

The difference is to Adam as above (from the paper)?

Algorithm 2 Adam with L_2 regularization and Adam with decoupled weight decay (AdamW)

given $\alpha = 0.001, \beta_1 = 0.9, \beta_2 = 0.999, \epsilon = 10^{-8}, \lambda \in \mathbb{R}$
initialize time step $t \leftarrow 0$, parameter vector $\boldsymbol{\theta}_{t=0} \in \mathbb{R}^n$, first moment vector $\boldsymbol{m}_{t=0} \leftarrow \boldsymbol{0}$, second moment vector $\boldsymbol{v}_{t=0} \leftarrow \boldsymbol{0}$, schedule multiplier $\eta_{t=0} \in \mathbb{R}$
repeat
  $t \leftarrow t+1$
  $\nabla f_t(\boldsymbol{\theta}_{t-1}) \leftarrow \text{SelectBatch}(\boldsymbol{\theta}_{t-1})$ ▷ select batch and return the corresponding gradient
  $\boldsymbol{g}_t \leftarrow \nabla f_t(\boldsymbol{\theta}_{t-1})$ $+\lambda\boldsymbol{\theta}_{t-1}$
  $\boldsymbol{m}_t \leftarrow \beta_1 \boldsymbol{m}_{t-1} + (1-\beta_1)\boldsymbol{g}_t$ ▷ here and below all operations are element-wise
  $\boldsymbol{v}_t \leftarrow \beta_2 \boldsymbol{v}_{t-1} + (1-\beta_2)\boldsymbol{g}_t^2$
  $\hat{\boldsymbol{m}}_t \leftarrow \boldsymbol{m}_t/(1-\beta_1^t)$ ▷ β_1 is taken to the power of t
  $\hat{\boldsymbol{v}}_t \leftarrow \boldsymbol{v}_t/(1-\beta_2^t)$ ▷ β_2 is taken to the power of t
  $\eta_t \leftarrow \text{SetScheduleMultiplier}(t)$ ▷ can be fixed, decay, or also be used for warm restarts
  $\boldsymbol{\theta}_t \leftarrow \boldsymbol{\theta}_{t-1} - \eta_t \left(\alpha \hat{\boldsymbol{m}}_t/(\sqrt{\hat{\boldsymbol{v}}_t}+\epsilon) +\lambda\boldsymbol{\theta}_{t-1}\right)$
until *stopping criterion is met*
return optimized parameters $\boldsymbol{\theta}_t$

So the difference is:

$$\begin{aligned} w_{t+1} = &\quad w_t - \eta_t \frac{m_t / c_{t,1}}{\sqrt{d_t / c_{t,2}} + \epsilon} \\ \text{vs } w_{t+1} = &\quad w_t - \eta_t \frac{m_t / c_{t,1}}{\sqrt{d_t / c_{t,2}} + \epsilon} - \lambda \eta_t w_t \end{aligned}$$

Many toolboxes do not do real weight decay, but add a ℓ_2-regularizer term and let the gradient perform implicitly weight decays then (see purple).

When implemented as ℓ_2-regularizer, then the effect of the ℓ_2-regularizer gets swallowed and smoothed out in/by the EMA terms.

EMA was designed to smooth out large changes in gradient, now it smoothens out the weight decay effect too :) .

> **Important:**
>
> AdamW compared to Adam performs a stronger weight decay when gradients are larger.

2 How valuable are these methods?

There are doubts that they are always better – you need to validate:

`https://arxiv.org/pdf/1705.08292.pdf`

Out of class:

My personal observation is that Adam converges faster in the beginning but SGD catches up later on. It seems that switching later to SGD can be beneficial: `https://arxiv.org/pdf/1712.07628.pdf`

> **Important:**
>
> If you want to use different solvers, remember to save not only the model but also the solver state

3 Where are those in pytorch?