50.021 -AI

Alex

Week 03: 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.]

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 $f_w : x \mapsto f(x) \in \mathcal{Y}$

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

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. 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`

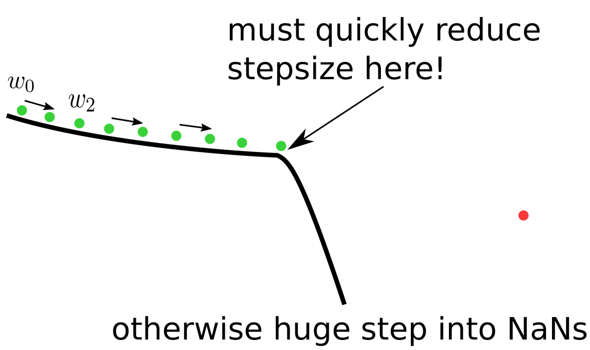
- DIVERGENCE if learning rate too high - (see example in past lecture)
- in a flat region steps can be very small:

  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: 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, (but in deep learning often too fast decrease of η_t)

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

2. *Batch gradient descent versus minibatch-gradient descent*

The idea in mini-batch gradient descent: at each time step use only k samples. Suppose until step $t + 1$ we have seen the first $N(t)$ samples already. Then at step $t+1$ we use the gradient over samples: $N(t)+1, N(t)+2, \ldots, N(t)+k$

$$\begin{aligned} w_{t+1} &= w_t - \eta_t \nabla_w \hat{E}(w_t, L, N(t)+1, N(t)+k) \\ &= w_t - \eta_t \nabla_w \sum_{i=N(t)+1}^{N(t)+k} L(f_w(x_i), y_i) \end{aligned}$$

Remarks

- usually number of steps t so big, that the whole dataset is gone through multiple times by minibatches.
- batch-size has an effect on memory usage - limiting with GPUs - because for a batch size of n usually n copies of the neural network are created in parallel and run in parallel.
- tradeoff: too small batch size too noisy, too large batch size: out of mem, or overfitting
- can help: every time when one starts from the first sample again, permutate order of samples for traversal by minibatches. Randomization of the order of samples can help to jump into good local minima. Randomization as strategy against overfitting.

3. *Weight decay*

Replace

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

by

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

shrinks weight towards zero. Comes from quadratic regularization:

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

So: weight decay is quadratic regularization.

one more constant for tunning

4. *Momentum term* → like real life momentum

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

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

- 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 smoothing
- with one more parameter α → need to tune
- 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

What does the momentum compute? Assume $\eta_t = \eta$ is constant. Lets shorten: $g_t = \nabla_w \hat{E}(w_t, L)$

$$m_1 = \alpha m_0 + \eta g_0 = \eta g_0$$
$$m_2 = \alpha m_1 + \eta g_1 = \alpha^1 \eta g_0 + \eta g_1$$
$$m_3 = \alpha m_2 + \eta g_2 = \alpha^2 \eta g_0 + \alpha^1 \eta g_1 + \eta g_2$$
$$m_4 = \alpha m_3 + \eta g_3 = \alpha^3 \eta g_0 + \alpha^2 \eta g_1 + \alpha^1 \eta g_2 + \eta g_3$$
$$m_5 = \alpha m_4 + \eta g_4 = \alpha^4 \eta g_0 + \alpha^3 \eta g_1 + \alpha^2 \eta g_2 + \alpha^1 \eta g_3 + \underset{\alpha^0}{\eta} g_4$$

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.
  of a time series

- 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): standard mean
$w_s = \frac{1}{t}$. This satisfies $\sum_{s=0}^{t-1} w_s = \sum_{s=0}^{t-1} \frac{1}{t} = 1$

For the momentum term:

$$\alpha^{t-1-s} \geq 0$$

weights for the past decrease exponentially

$$\sum_{s=0}^{t-1} \alpha^{t-1-s} = \alpha^{t-1} + \alpha^{t-2} + \alpha^{t-3} + \ldots + \alpha^2 + \alpha^1 + \alpha^0$$
$$= \sum_{s=0}^{t-1} \alpha^s = \frac{1-\alpha^t}{1-\alpha}$$

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.

5. *Exponential moving average (EMA)*

For a time series g_s the term

similer to momentum term

$$EMA(g_s, s \leq 0) = 0 + (1-\alpha) g_0$$
$$EMA(g_s, s \leq t) = \alpha EMA(g_s, s \leq t-1) + (1-\alpha) g_t$$

defines an exponential moving average. Moving – because weights are high for recent past.

The recursion yields here

$$EMA(g_s, s \le 0) = (1-\alpha)g_0$$
$$EMA(g_s, s \le 1) = \alpha^1(1-\alpha)g_0 + (1-\alpha)g_1$$
$$EMA(g_s, s \le 2) = \alpha^2(1-\alpha)g_0 + \alpha^1(1-\alpha)g_1 + (1-\alpha)g_2$$
$$EMA(g_s, s \le 3) = \alpha^3(1-\alpha)g_0 + \alpha^2(1-\alpha)g_1 + \alpha^1(1-\alpha)g_2 + (1-\alpha)g_3$$
$$EMA(g_s, s \le t) = \sum_{s=0}^{t} \alpha^{t-s}\underbrace{(1-\alpha)}\underbrace{g_s}$$

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

6. 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:

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

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

$$w_{t+1} = w_t - \eta_t \frac{g_t}{\|g_t\|} \quad \text{for larger step size}$$

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 EMA of norms of gradients:

$$w_{t+1} = w_t - \eta_t \frac{g_t}{EMA(\|g_s\|, s \le t)}$$

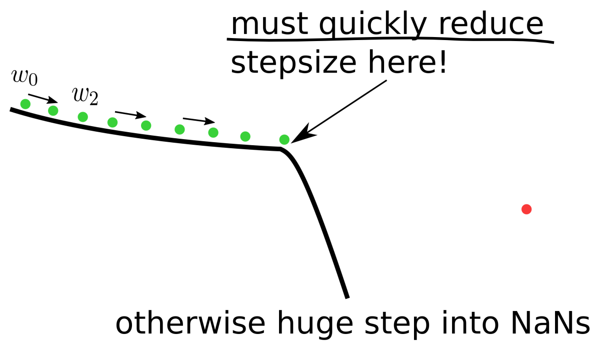
Idea: flat valley, for many time steps s around the current time step t norms of gradients are small, so EMA will be small. Dividing by a small term makes the stepsize bigger.

look to the past

Still not perfect: We need to reduce the stepsize fast when we enter more steep regions. That means: if a current gradient norm $\|g_t\|$

but need to look ahead

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).

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

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

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

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:

$$\text{compute } g_t := \nabla_w \hat{E}(w_t, L)$$
$$EMA(\|g_s\|^2, s \leq t) = \alpha EMA(\|g_s\|^2, s \leq t-1) + (1-\alpha)\|g_t\|^2$$
$$w_{t+1} = w_t - \eta_t \frac{g_t}{\sqrt{EMA(\|g_s\|^2, s \leq t) + \epsilon}}$$

square root of square

RMSPROP coarse idea:

- 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

Math is thinking in patterns

* arithmetic mean

$$M(g_i) := \left(\frac{1}{N}\sum_{i=1}^{N} g_i\right)^{\frac{1}{1}}$$

* harmonic mean

$$H(g_s) = \left(\frac{1}{N}\sum_{s=1}^{N} (g_s)^{-1}\right)^{-\frac{1}{1}}$$

* Quadratic mean

$$Q(g_s) := \left(\frac{1}{N}\sum_{s=1}^{N} g_s^{2}\right)^{\frac{1}{2}}$$

⋮

larger.

more sensitive to large outliers

6. *AdaDelta*

Adadelta starts with an observation from RMSPROP – the physical units are wrong. If w has a physical unit (eg meters), then the update should have the same physical unit (as it is a direction in the parameter space).

the RMSprop update is unitless:

$$\begin{aligned} w_{t+1} &= w_t - \eta_t \frac{g_t}{\sqrt{EMA(\|g_s\|^2, s \leq t) + \epsilon}} \\ meters &= m - \frac{m}{\sqrt{m^2}} = meters - unitless \end{aligned}$$

so must correct it by adding some new hack. Took inspiration: second order optimization methods (using the Hessian matrix – the matrix of second derivatives).

$$\boxed{w_{t+1} = w_t - \eta_t \frac{g_t}{\sqrt{EMA(\|g_s\|^2, s \leq t) + \epsilon_1}} \sqrt{EMA(\|w_{s+1} - w_s\|^2, s \leq t-1) + \epsilon_2}}$$

Note here: $EMA(\|w_{s+1} - w_s\|^2, s \leq t-1)$ needs only knowledge up to w_t, so no recursion :) .

Quick explanation (**out of lecture!** ... better read the paper): Second order updates. Taylor expansion yields

$$w_{t+1} = w_t + \Delta w$$
$$E(w_t + \Delta w) \approx E(w_t) + \Delta w \nabla_w E(w_t) + \frac{1}{2}\Delta w^T H \Delta w$$
$$H_{ik} = \frac{\partial^2 E}{\partial w^{(i)} \partial w^{(k)}}(w = w_t)$$

Minimum of $E(w_t + \Delta w)$ as a function of update Δw is:

$$0 = \nabla_{(\Delta w)} E(w_t + \Delta w) \approx 0 + \nabla_w E(w_t) + \Delta w^T H$$
$$\Delta w = H^{-1} \nabla_w E(w_t)$$

In this update the units are correct:

H can be seen as the derivative of $\nabla_w E$ with respect to w, so the units of H are $\frac{unit\ of\ E}{(unit\ of\ w)^2}$.

The units of an inverse are the inversed units, so the units of H^{-1} are $\frac{(unit\ of\ w)^2}{unit\ of\ E}$.

The unit of $\nabla_w E(w_t)$ is $\frac{unit\ of\ E}{(unit\ of\ w)}$.

so the units of the update are $\frac{(unit\ of\ w)^2}{unit\ of\ E} \frac{unit\ of\ E}{(unit\ of\ w)} = unit\ of\ w$.

For AdaDelta:
$R = \frac{1}{\sqrt{EMA(\|\nabla_w \hat{E}(w_s, L)\|^2)_{s=0}^{t} + \epsilon}} \sqrt{EMA((\|w_{s+1} - w_s\|^2)_{s=0}^{t-1}) + \epsilon}$ has the same units as H^{-1}.

$\sqrt{EMA(\|\nabla_w \hat{E}(w_s, L)\|^2)_{s=0}^{t} + \epsilon}$ has units $\sqrt{\frac{(unit\ of\ E)^2}{(unit\ of\ w)^2}}$.

$\sqrt{EMA((\|w_{s+1} - w_s\|^2)_{s=0}^{t-1}) + \epsilon}$ has units $\sqrt{(unit\ of\ w)^2}$, so R has units $\frac{(unit\ of\ w)^2}{unit\ of\ E}$ – which are the same units as H^{-1}.

Multiplying the units for $\nabla_w E \cdot R$ shows that it has units $unit\ of\ w$ – therefore this term is suitable for an update

7. Adam

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?

$$
\begin{aligned}
\text{compute } g_t &:= \nabla_w \hat{E}(w_t, L) \\
EMA(\|g_s\|^2, s \leq t) &= \alpha_1 EMA(\|g_s\|^2, s \leq t-1) + (1-\alpha_1)\|g_t\|^2 \\
rpropterm &= \frac{g_t}{\sqrt{EMA(\|g_s\|^2, s \leq t) + \epsilon}}
\end{aligned}
$$

in RMSProp one would apply *rpropterm* to update the weights w_t with a stepsize η_t. Now comes the momentum computation (a bit differently written from above momentum term equation, but it is the same up to a constant)

$$
\begin{aligned}
EMA(m_s, s \leq t) &= \alpha_2 EMA(m_s, s \leq t-1) + (1-\alpha_2) rpropterm \\
w_{t+1} &= w_t - \eta_t EMA(m_s, s \leq t)
\end{aligned}
$$

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

1. normalize every dimension of the update separately
2. turn all used/defined terms which use an EMA into a true weighted average by multiplying them with an appropriate constant

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:

$$\frac{1}{\sqrt{EMA(\|g_s\|^2, s \leq t) + \epsilon}}$$

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

$$EMA((g_s^{(d)})^2, s \leq t) = \alpha_1 EMA((g_s^{(d)})^2, s \leq t-1) + (1-\alpha_1)(g_t^{(d)})^2$$

The gradient gets normalized in every dimension d separately:

$$term^{(d)} = \frac{g_t^{(d)}}{\sqrt{EMA((g_s^{(d)})^2, s \leq t) + \epsilon}}$$

Using only 1. the algorithm would look like that:

$$\begin{aligned}
\text{compute } g_t &:= \nabla_w \hat{E}(w_t, L) \\
EMA((g_s^{(d)})^2, s \leq t) &= \alpha_1 EMA((g_s^{(d)})^2, s \leq t-1) + (1-\alpha_1)(g_t^{(d)})^2 \\
term^{(d)} &= \frac{g_t^{(d)}}{\sqrt{EMA((g_s^{(d)})^2, s \leq t) + \epsilon}} \\
EMA(m_s, s \leq t) &= \alpha_2 EMA(m_s, s \leq t-1) + (1-\alpha_2) term \quad \text{(for every dimension } d\text{)} \\
w_{t+1} &= w_t - \eta_t EMA(m_s, s \leq t)
\end{aligned}$$

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 $EMA(u_s, s \leq t)$ 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:

$$
\begin{aligned}
\text{compute } g_t &:= \nabla_w \hat{E}(w_t, L) \\
EMA((g_s^{(d)})^2, s \le t) &= \alpha_1 EMA((g_s^{(d)})^2, s \le t-1) + (1-\alpha_1)(g_t^{(d)})^2 \\
divEMA_1 &= \frac{EMA((g_s^{(d)})^2, s \le t)}{1-\alpha_1^{t+1}} \\
term^{(d)} &= \frac{g_t^{(d)}}{\sqrt{divEMA_1 + \epsilon}} \\
EMA(m_s, s \le t) &= \alpha_2 EMA(m_s, s \le t-1) + (1-\alpha_2) term \quad \text{(for every dimension } d\text{)} \\
divEMA_2 &= \frac{EMA(m_s, s \le t)}{1-\alpha_2^{t+1}} \\
w_{t+1} &= w_t - \eta_t divEMA_2
\end{aligned}
$$

} true weighted average

(sum up to 1; weights nonnegative)