IN5400 – Optimizers, a bit of modern neural nets for classification

Alexander Binder

February 27, 2022

Key takeaways for Optimizers beyond SGD

Be able to explain the main ideas behind:

- learning rate decay strategies
- learning rate: linear warmup
- weight decay and its equivalence to ℓ_2-regularization
- the concept of exponential moving averages of a time-series
- updates in momentum term, RMSprop, Adam use exponential moving averages (EMA)
- momentum term
- RMSprop
- Adam
- AdamW

Key takeaways for Modern neural net components

- squeeze-and-excitation blocks as plugin module (method)
- resnext (arch) and grouped convolutions (method)
- normalizer-free resnets (arch) and scaled weight standardization (method)
- normalizer-free nets (arch) and gradient (method)

1 Optimizers beyond SGD: Better way to update parameters by applying gradients

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

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

$$\text{argmin}_w \hat{E}(f_w, L)$$

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:
  - $w_{t+1} = w_t - \eta \nabla_w \hat{E}(w_t, L) = w_t - \text{learningrate} \cdot \frac{dE}{dw}(w_t)$
  - compute change to last: $\|w_{t+1} - w_t\|$

Problem is: deep neural networks have many parameters - see code. 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.

fixed learning rate can result in problems: `quadform.py`

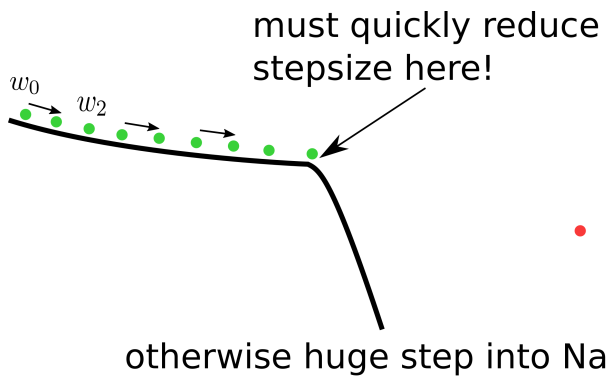
- DIVERGENCE if learning rate too high - (see example in past lecture)
- in a flat region steps can be very small:

  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:

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

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

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

1.2 Linear learning rate warm up

Goyal et al. `https://arxiv.org/abs/1706.02677` inspired by Chen et al. `https://ieeexplore.ieee.org/document/7472805`. Section 2.2 in the former

$$\eta_t = \begin{cases} \eta_0 \frac{t}{K_0+1} & \text{if } t \leq K_0, \text{ warmup phase} \\ f(t, \eta_0, K_0) & \text{if } t > K_0, \ f(\cdot) \text{ is your usual learning rate decay,} \\ & \text{offsetted to start at index } K_0 \end{cases}$$

Motivation: resolve instabilities at the start of learning by starting slower during the first K_0 epochs, in `https://arxiv.org/abs/1706.02677` $K_0 = 5$ for imagenet. Might need to be larger for smaller datasets.

Results: see Table 1 in the paper `https://arxiv.org/abs/1706.02677`.

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-5f511975163`

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

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

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

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

1.3 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:

$$\begin{aligned} \hat{E}_{Reg}(w, L) &= \frac{1}{n}\sum_{i=1}^{n} L(f_w(x_i), y_i) && +\frac{1}{2}\lambda\eta_t\|w\|_2^2 \\ \nabla_w \hat{E}_{Reg}(w, L) &= \nabla_w \frac{1}{n}\sum_{i=1}^{n} L(f_w(x_i), y_i) && +\nabla_w \frac{1}{2}\lambda\eta_t\|w\|_2^2 \\ \nabla_w \hat{E}_{Reg}(w, L) &= \nabla_w \hat{E}(w, L) && +\lambda\eta_t w \end{aligned}$$

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

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

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 $w = w \cdot (1 - \gamma), \gamma \in (0, 1)$ to shrink it towards zero.

1.4 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.

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

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:

$$
\begin{gathered}
\sum_{s=0}^{t-1} w_s g_s \\
w_s \geq 0, \ \sum_{s=0}^{t-1} w_s = 1
\end{gathered}
$$

Vanilla average is a weighted average with constant (time-independent weights): $w_s = \frac{1}{t}$.

For the momentum term:

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

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

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

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, 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.

- momentum does what?: 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 remembers the bigger stepsize from the past steps
- with one more parameter α

Important:

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

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

Momentum term can be understood as a rescaled, time-dependent weighted average of gradients. Meaning of Rescaled: weights do not sum up to 1 and depend on the number of steps T. Momentum is related to first moment of gradients and thus to the idea of replacing a random variable by its first moment.

pytorch ?

```
optimizer = torch.optim.SGD(model.parameters(), lr=lr, momentum=0.9)
```

1.5 Nesterov momentum

Recap momentum:

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

Can rewrite last update w_{t+1} as:

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

Largest component of it is usually the old weight and the momentum term $w_t - \alpha m_t$. Meaning: approximate next position is $w_t - \alpha m_t$.
Idea of Nesterov: compute gradient at the approximate next position $w_t - \alpha m_t$:

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

Nesterov momentum:

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

1.6 Exponential moving average (EMA)

For a time series g_s the term

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

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

Generalize what we have seen in the momentum. Difference: explicit weighting with $1-\alpha$.

The recursion yields here

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

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

1.7 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:

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

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 into the past.

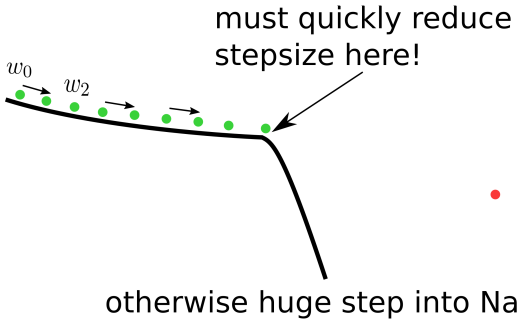
So use an average of norms of gradients from the past, and divide by them. Divide by $\text{EMA}(\cdot)_t$ of norms of gradients:

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

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)!

Need to make the EMA more sensitive to large gradient in the current time 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).

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

One can show mathematically that the root of an average of squared norms is larger than the vanilla average: inequality between the arithmetic and the quadratic mean.

Still not perfect: What is if all gradients are near-zero? Huge step into the world of NaNInf. Better: add a small ϵ

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

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

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

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

RMSProp can be understood as a normalization of the gradient by a rescaled, time-dependent weighted average of gradient norms. The normalizer in RMSProp is related to the second moment of gradient norms. Compare also to standard deviation. The normalizer in RMS prop differs from standard deviation in what way ?

$$\begin{aligned} X &\text{ random variable} \\ E[X] &\text{ first moment} \\ E[X^2] &\text{ second moment} \\ \sigma^2(X) &= E[X^2] - (E[X])^2 \end{aligned}$$

It is related to normalizing a variable by dividing it by an estimate of its standard deviation.

$$X \longrightarrow \frac{X}{\sqrt{\sigma^2(X) + \epsilon}}$$

Compare to RMSprop:

$$\begin{aligned} Z &= g_t \\ N &= EMA(\|g_t\|^2) \text{ rescaled second moment estimator for } \|g_t\| \\ u_t &= \frac{Z}{\sqrt{N + \epsilon}} = \frac{g_t}{\sqrt{EMA(\|g_t\|^2) + \epsilon}} \\ w_{t+1} &= w_t - \eta_t u_t \end{aligned}$$

RMSprop does something like:

$$X_d \longrightarrow \frac{X_d}{\sqrt{E[\|X\|^2] + \epsilon}}$$

How to express a similar intuition for the momentum?

Compare this to momentum where (The EMA is not an exact first moment and not an exact expectation!!) the idea was:

$$X \longrightarrow E[X]$$

Putting these two ideas together will result in Adam

$$X \longrightarrow \frac{E[X]}{\sqrt{E[X^2] + \epsilon}}$$

1.8 Adam

A popular 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?

$$\begin{aligned} &\text{compute } g_t := \nabla_w \hat{E}(w_t, L) \\ &\qquad s_t = \alpha_1 s_{t-1} + (1-\alpha_1)\|g_t\|^2 \quad \#\ d_t\ is\ EMA(\|g_s\|^2)_t \\ &rpropterm = \frac{g_t}{\sqrt{s_t}+\epsilon} \end{aligned}$$

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:

$$\begin{aligned} m_t &= \alpha_2 m_{t-1} + (1-\alpha_2) g_t \\ w_{t+1} &= w_t - \eta_t \frac{m_t}{\sqrt{s_t}+\epsilon} \end{aligned}$$

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 normalizer (time-dependent) $\frac{1}{1-\alpha^t}$

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:

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

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 dimension d:

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

$s_t[d]$ is a scalar. Note summing the component squared for all dimensions d results in the squared gradient norm $\|g_t\|^2 = \sum_d (g_t[d])^2$.

For a single dimension of the gradient this follows the idea of replacing a random variable with its mean, and to normalize it by an estimate of the $\sqrt{\cdot}$ of second moment which is related to the standard deviation:

$$g_t[d] \longrightarrow \frac{E[\, g_t[d]\,]}{\sqrt{E[\, g_t[d]^2\,]} + \epsilon}$$

It does not hold exactly bcs the weights of the EMA do not sum up to one

Using only 1. the algorithm would look like that:

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

$$\begin{aligned} s_t[d] &= \alpha_1 s_{t-1} + (1-\alpha_1)(g_t[d])^2 \ \#\text{EMA of } g_t[d]^2 \\ s_t &= (s_t[0], \ldots, s_t[D]) = (s_t[d])_d \ \#\text{vectorization} \\ m_t &= \alpha_2 m_{t-1} + (1-\alpha_2) g_t \ \#\text{EMA of } g_t[d]\text{vectorized} \\ w_{t+1} &= w_t - \eta_t \frac{m_t}{\sqrt{s_t} + \epsilon} \end{aligned}$$

$1/\sqrt{s_t}$: (element-wise division for every dimension $s_t[d]$)

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

$$m_0 = 0, s_0 = 0$$

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

$$s_t[d] = \alpha_1 s_{t-1}[d] + (1-\alpha_1)(g_t[d])^2 \quad \#element-wise$$

$$m_t = \alpha_2 m_{t-1} + (1-\alpha_2) g_t$$

$$c_{t,1} = 1 - \alpha_1^t, c_{t,2} = 1 - \alpha_2^t$$

$$w_{t+1} = w_t - \eta_t \frac{m_t / c_{t,1}}{\sqrt{s_t / c_{t,2}} + \epsilon}$$

$1/\sqrt{s_t}$: (element-wise division for every dimension $s_t[d]$)

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 $+\epsilon$.

1.9 AdamW: Adam with decoupled weight decay

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

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

$$m_0 = 0, s_0 = 0$$

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

$$s_t[d] = \alpha_1 s_{t-1}[d] + (1-\alpha_1)(g_t[d])^2 \quad \#element-wise$$

$$m_t = \alpha_2 m_{t-1} + (1-\alpha_2) g_t$$

$$c_{t,1} = 1 - \alpha_1^t, c_{t,2} = 1 - \alpha_2^t$$

$$w_{t+1} = w_t - \eta_t \frac{m_t / c_{t,1}}{\sqrt{s_t / c_{t,2}} + \epsilon} - \lambda \eta_t w_t$$

$1/\sqrt{s_t}$: (element-wise division for every dimension) $s_t[d]$)

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:

$$w_{t+1} = w_t - \eta_t \frac{m_t / c_{t,1}}{\sqrt{d_t / c_{t,2}} + \epsilon}$$

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

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.

1.10 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

1.11 Where are those in pytorch?

`torch.optim`

1.12 A paper on effects of newer learning rate heuristics

`https://arxiv.org/pdf/1810.13243.pdf`

Why all these arxiv.org papers

Learning from lectures ??? Learn to effectively process/filter papers: ICML, NeurIPS, ICLR, CVPR, ICCV, ACL, EMNLP, ICASSP, SIGGRAPH, ... and rank A and rank B conferences.

2 SOA 2019/2020/2021

- squeeze-and-excitation blocks as plugin module (method)
- resnext (arch) and grouped convolutions (method)
- normalizer-free resnets (arch) and scaled weight standardization (method)
- normalizer-free nets (arch) and gradient (method)

2.1 squeeze-and-excitation blocks as plugin module

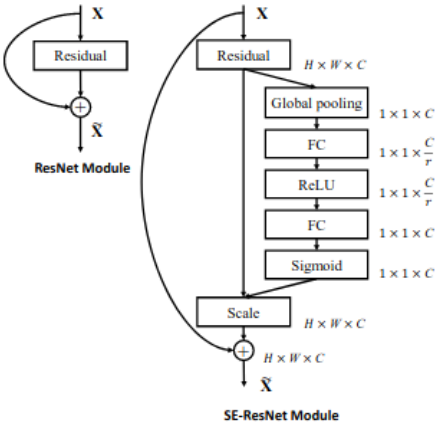


Fig. 3. The schema of the original Residual module (left) and the SE-ResNet module (right).

idea behind it:

- have feature map $x_{l+1}[b, c, h, w]$ computed as shortcut plus residual block f_l.

$$x_{l+1} = x_l + f_l(x_l)$$
$$x_{l+1}[b, c, h, w] = x_l[b, c, h, w] + f_l[b, c, h, w]$$

  Goal: Want to compute a reweighting of each channel c in the residual block $f_l[b, c, h, w]$.

- reweighting should depend on statistic computed from all channels, aggregated over all spatial components h, w. How to design that ?

- first: pool feature map across spatial components h, w:

$$z[b, c] = \text{Pool}(f_l[b, c, :, :])$$

  This gives a vector $z_b = (z[b, c])_{c=1}^{C}$ with shape $(channels)$ for each batch element b. Next: let channels interact with each other by two fully-connected layers with a relu in between:

$$s_b = \sigma(W_2 \text{ReLU}(W_1 z_b))$$

- s learns a reweighting by letting channels interact with each other
- multiply output of residual block with the reweighting:

$$\tilde{f}_l = s[:, c] f_l[:, c, h, w]$$

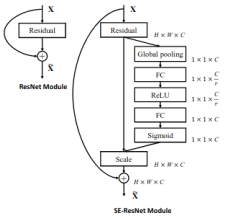


Fig. 3. The schema of the original Residual module (left) and the SE-ResNet module (right).

- then add shortcut on top of channel-weighted residual

$$\begin{aligned} x_{l+1}[b, c, h, w] &= x_l[b, c, h, w] + \tilde{f}_l[b, c, h, w] \\ &= x_l[b, c, h, w] + s[:, c] f_l[:, c, h, w] \end{aligned}$$

- light-weight module, s weights do not sum to 1, component-wise rescaling rather than attention weights
- note the bottleneck C/r in the Figure above

2.2 From ResNet to ResNext

Xie et al. `https://arxiv.org/abs/1611.05431`

- use bottlenecks like in the larger resnets

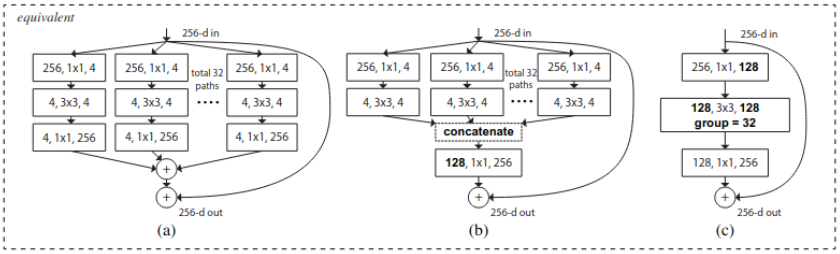


Figure 3. Equivalent building blocks of ResNeXt. **(a)**: Aggregated residual transformations, the same as Fig. 1 right. **(b)**: A block equivalent to (a), implemented as early concatenation. **(c)**: A block equivalent to (a,b), implemented as grouped convolutions [24]. Notations in **bold** text highlight the reformulation changes. A layer is denoted as (# input channels, filter size, # output channels).

The original resnet bottleneck for resnet layer1 would be subfigure (c) without the $C = 32$ groups, thus having a kernel weight parameter of shape $(128, 3, 3, 128)$ which processes all 128 input channels and maps them onto 128 output channels.
The resnext, here shown for layer1, uses a 3×3 convolution with 32 groups, that is, it has 32 separate kernels of size $128/32, 3, 3, 128/32$. Each of the separate kernels takes $128/32 = 4$ input channels and maps them onto 4 output channels. In total all 128 input channels are mapped onto 128 output channels.

- use grouped convolutions for the 3×3 convolutions in a bottleneck block

stage	output	ResNet-50	**ResNeXt-50 (32×4d)**
conv1	112×112	7×7, 64, stride 2	7×7, 64, stride 2
conv2	56×56	3×3 max pool, stride 2	3×3 max pool, stride 2
		[1×1, 64 3×3, 64 1×1, 256] ×3	[1×1, 128 3×3, 128, C=32 1×1, 256] ×3
conv3	28×28	[1×1, 128 3×3, 128 1×1, 512] ×4	[1×1, 256 3×3, 256, C=32 1×1, 512] ×4
conv4	14×14	[1×1, 256 3×3, 256 1×1, 1024] ×6	[1×1, 512 3×3, 512, C=32 1×1, 1024] ×6
conv5	7×7	[1×1, 512 3×3, 512 1×1, 2048] ×3	[1×1, 1024 3×3, 1024, C=32 1×1, 2048] ×3
	1×1	global average pool 1000-d fc, softmax	global average pool 1000-d fc, softmax
# params.		**25.5**×10^6	**25.0**×10^6
FLOPs		**4.1**×10^9	**4.2**×10^9

Table 1. (**Left**) ResNet-50. (**Right**) ResNeXt-50 with a 32×4d template (using the reformulation in Fig. 3(c)). Inside the brackets are the shape of a residual block, and outside the brackets is the number of stacked blocks on a stage. "C=32" suggests grouped convolutions [24] with 32 groups. *The numbers of parameters and FLOPs are similar between these two models.*

2.3 the predecessor variant: normalizer-free resnets

Brock et al. `https://arxiv.org/abs/2101.08692`

- resnet base architecture with changes. Start is resnet block

$$x_{l+1} = x_l + f_l(x_l)$$

- step 1: remove Batchnorm.
- step 2 , **key idea 1:** replace shortcut with residual branch

$$x_{l+1} = x_l + f_l(x_l)$$

by a variant which allows to rescale variances:

$$x_{l+1} = x_l + \alpha f_l(x_l/\beta_l)$$

remember:

$$Var(ax) = a^2 Var(x),\ Std(ax) = aStd(x)$$

- **key idea 2:** preserve variance across layers at initialization of the network so that $Var(x_{l+1}) \approx Var(x_l) + \alpha^2$, where α is a learnable/trainable constant.

(d.1) in particular, make sure that for the residual branch f_l the following holds $Var(f_l(z)) = Var(z)$ – **scaled weight standardization** (further down below) will make that happen.

- set $\beta_l = Var(x_l)$, then by (d.1)

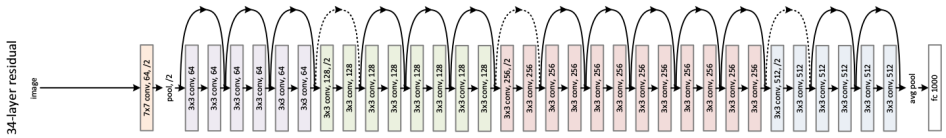
$$Var(f_l(x_l/\beta_l)) = Var(x_l\ \beta_l) = 1$$

as a consequence:

$$Var(x_{l+1}) \approx Var(x_l) + \alpha^2$$

as desired.

- next: modification to transition layers. A transition layer is where the resolution is reduced.



- at transition layers, use as input x_l/β_l, so that the Variance of the feature map at transition layers becomes 1. The same effect happens in transition layers where the feature map size is changing. In such layers there cannot be a shortcut, and for conventional resnets $conv1x1$ stride 2 with batchnorm or similar operators are used.

- in order to make (d.1) happen: instead of batchnorm use scaled weight standardization, without group norm

$$\widehat{W}_{o,i} = \gamma \frac{W_{o,i} - \mu_{W[o,:]}}{\sqrt{N}\sigma_{W[o,:]}}$$

$$\mu_{W[o,:]} = \frac{1}{N}\sum_i W_{o,i}$$

$$\sigma^2_{W[o,:]} = \frac{1}{N}\sum_i (W_{o,i} - \mu_{W[o,:]})^2$$

where N are the fan-in, = the number of inputs used to compute the mean. Example: Convolution with kernel 3×3, 256 in channels, 128 out channels has a fan-in of $3 \cdot 3 \cdot 256$.

Why? Consider a linear operator (e.g. convolution) after a relu $g(\cdot)$

$$z = Wg(x)$$

$$z_o = \sum_{i=1}^{N} W_{o,i} g(x_i)$$

If $\mathbb{E}[g(x)] = \mu_g$ and $Var(g(x)) = \sigma_g^2$, then

$$\mu(z_o) = N\mu_g\mu_{W[o,:]}$$
$$Var(z_o) = N\sigma_g^2(\sigma_{W[o,:]}^2 + \mu_{W[o,:]}^2)$$

Using scaled weight standardization we obtain, obviously $\mu_{\widehat{W}[o,:]} = 0$ and furthermore:

$$\sigma_{\widehat{W}[o,:]}^2 = \gamma^2/N$$

This implies then:

$$\mu(z_o) = 0$$
$$Var(z_o) = N\sigma_g^2(\sigma_{\widehat{W}[o,:]}^2 + 0) = \gamma^2\sigma_g^2$$

Next they calculate σ_g^2 for a relu unit and standard normal inputs. This can be computed (rectified gaussian distribution) as $\frac{1}{2}(1 - \frac{1}{\pi})$. This tells how to set γ , namely $\gamma = \sqrt{\frac{1}{\sigma_g^2}}$ which is:

$\gamma = \sqrt{\frac{2}{1-\frac{1}{\pi}}}$

Summary: using an initialization, such that each relu gets standard normal inputs, relu networks and using scaled weight standardization with the above γ, they obtain property (d.1) $Var(f_l(z)) = Var(z)$ at initialization time

5.1 An empirical evaluation on ResNets

	FixUp		SkipInit		NF-ResNets (ours)		BN-ResNets	
	Unreg.	Reg.	Unreg.	Reg.	Unreg.	Reg.	Unreg.	Reg.
RN50	$74.0 \pm .5$	$75.9 \pm .3$	$73.7 \pm .2$	$75.8 \pm .2$	$75.8 \pm .1$	$\mathbf{76.8 \pm .1}$	$\mathbf{76.8 \pm .1}$	$76.4 \pm .1$
RN101	$75.4 \pm .6$	$77.6 \pm .3$	$75.1 \pm .1$	$77.3 \pm .2$	$77.1 \pm .1$	$\mathbf{78.4 \pm .1}$	$78.0 \pm .1$	$78.1 \pm .1$
RN152	$75.8 \pm .4$	$78.4 \pm .3$	$75.7 \pm .2$	$78.0 \pm .1$	$77.6 \pm .1$	$\mathbf{79.1 \pm .1}$	$78.6 \pm .2$	$78.8 \pm .1$
RN200	$76.2 \pm .5$	$78.7 \pm .3$	$75.9 \pm .2$	$78.2 \pm .1$	$77.9 \pm .2$	$\mathbf{79.6 \pm .1}$	$79.0 \pm .2$	$79.2 \pm .1$
RN288	$76.2 \pm .4$	$78.4 \pm .4$	$76.3 \pm .2$	$78.7 \pm .2$	$78.1 \pm .1^*$	$\mathbf{79.5 \pm .1}$	$78.8 \pm .1$	$\mathbf{79.5 \pm .1}$

Table 1: ImageNet Top-1 Accuracy (%) for ResNets with FixUp (Zhang et al., 2019) or SkipInit (De & Smith, 2020), Normalizer-Free ResNets (ours), and Batch-Normalized ResNets. We train all variants both with and without additional regularization (stochastic depth and dropout). Results are given as the median accuracy $\pm$ the standard deviation across 5 random seeds. * indicates a setting where two runs collapsed and results are reported only using the 3 seeds which train successfully.

In the paper they derive how to compute constants γ for other activations, how to approximately adjust these γ for squeeze excitation modules.

- stochastic depth regularization: $p = 0.1$ probability to drop residual block f_l in

$$x_{l+1} = x_l + f_l(x_l)$$

- dropout before the final layer

- many other small tricks (section C of paper appendix, no need to memorize those)
  - modified squeeze and excite layers with correction for γ
  - at spatial transition layers: replace 1×1 convolution with stride 2 by average pooling with kernel size 1×1 followed by 1×1 convolution with stride 1
  - learning rate warmup in the first 5 epochs until it reaches 0.4
  - SGD with nesterov momentum, batchsize 1024
- able to omit Batchnorm, but performance below that of efficientnet architectures `https://arxiv.org/abs/1905.11946`

	NF-ResNets (ours)			BN-ResNets		
	BS=1024	BS=8	BS=4	BS=1024	BS=8	BS=4
ResNet-50	69.9 ± 0.1	$\mathbf{69.6 \pm 0.1}$	$\mathbf{69.9 \pm 0.1}$	$\mathbf{70.9 \pm 0.1}$	65.7 ± 0.2	55.7 ± 0.3

Table 2: ImageNet Top-1 Accuracy (%) for Normalizer-Free ResNets and Batch-Normalized ResNet-50s on ImageNet, using very small batch sizes trained for 15 epochs. Results are given as the median accuracy $\pm$ the standard deviation across 5 random seeds. Performance degrades severely for Batch-Normalized networks, while Normalizer-Free ResNets retain good performance.

- good at small batchsizes ≤ 1024

Some more steps needed to get better than efficientnets

2.4 the final variant: normalizer-free networks

Brock et al. `https://arxiv.org/abs/2102.06171`

- on top of the ideas from normalizer-free resnets
- one novel big idea: adds adaptive gradient clipping. General idea of standard gradient clipping is to keep gradient norms bounded

$$clip_\lambda(g) = \begin{cases} g & \text{if } \|g\| \leq \lambda \\ g\frac{\lambda}{\|g\|} & \text{if } \|g\| > \lambda \end{cases}$$

Motivation: ensures $\|g\| \leq \lambda$

proposed **adaptive gradient clipping:**

Consider one output neuron i at layer l - that is the value $x_l[b, c, h, w]$ for a feature map x_l, where the index i runs over all possible index combinations (b, c, h, w) combinations. It has a gradient vector G_l^i with respect to all its inputs used to compute $x_l[b, c, h, w]$. It has also a weight vector W_l^i used to compute $x_l[b, c, h, w]$. Then adaptive gradient clipping computes:

$$\|W\|_F^* = \max(\|W\|_F, \epsilon), \ \|W\|_F = \sum_{i,k} W_{ik}^2$$

$$adaclip_\lambda(g) = \begin{cases} G_l^i & \text{if } \frac{\|G_l^i\|}{\|W_l^i\|_F^*} \leq \lambda \\ G_l^i \frac{\lambda \|W_l^i\|_F^*}{\|G_l^i\|} & \text{if } \frac{\|G_l^i\|}{\|W_l^i\|_F^*} > \lambda \end{cases}$$

See Figure 2 in the paper for possible settings for λ.

Motivation? They argue: training becomes unstable when the gradient update scale relative to the scale of current parameters becomes large.

For standard SGD the update would be:

$$\theta_{t+1} - \theta_t = -\eta \nabla L(\theta_t)$$
$$\delta\theta_{|t,t+1} = -\eta \nabla L(\theta_t)$$

Therefore the gradient update scale relative to the scale of current parameters can be obtained as:

$$\frac{\|\delta\theta_{|t,t+1}\|}{\|\theta_t\|} = \eta \frac{\|\nabla L(\theta_t)\|}{\|\theta_t\|} \approx \eta \frac{\|G_l^i\|}{\|W_l^i\|_F^*}$$

Why $\frac{\|G_l^i\|}{\|W_l^i\|_F^*}$ and not $\frac{\|G\|}{\|W\|_F^*}$? One can reduce it to the question of:

Why $\|G_l^i\|$ and not $\|\nabla L(\theta_t)\|$?

$\nabla L(\theta_t)$ is the vector of all parameter gradients. Its norm $\|\nabla L(\theta_t)\|$ contains norms from all layers and from all neurons within a layer.
You can see it for a concatenation of 3 vectors v_i: if (v_1, v_2, v_3), then

$$\|(v_1, v_2, v_3)\|^2 = \sum_{d=1}^{D} v_{1,d}^2 + \sum_{d=1}^{D} v_{2,d}^2 + \sum_{d=1}^{D} v_{3,d}^2 \quad = \|v_1\|^2 + \|v_2\|^2 + \|v_3\|^2$$
$$\|(v_1, v_2, v_3)\| = \sqrt{\|v_1\|^2 + \|v_2\|^2 + \|v_3\|^2}$$

They argue: one should treat each output neuron separately for clipping purposes.

With respect to different layers: There is little reason to modify the gradient of one layer based on a statistic which includes other layers which are far away.

With respect to different neurons within the same layer: gradients can be locally different because an image can be locally highly varying. So clipping should be performed on separate neurons.

Therefore one looks at gradients computed G_l^i for layer l and output neuron i separately from other neurons and other layers. Same for weights: W_l^i.

- changes in the base model: replace SE-resnets by a SE-Resnext
- changed number of blocks in layer1 to layer4, with train and test image size depending on network complexity (Touvron et al `https://arxiv.org/abs/1906.06423`): Table 1 in the paper

Table 1. NFNet family depths, drop rates, and input resolutions.

Variant	Depth	Dropout	Train	Test
F0	[1, 2, 6, 3]	0.2	192px	256px
F1	[2, 4, 12, 6]	0.3	224px	320px
F2	[3, 6, 18, 9]	0.4	256px	352px
F3	[4, 8, 24, 12]	0.4	320px	416px
F4	[5, 10, 30, 15]	0.5	384px	512px
F5	[6, 12, 36, 18]	0.5	416px	544px
F6	[7, 14, 42, 21]	0.5	448px	576px

- added substantial data augmentation: RandAugment (last lecture) and Mixup `https://arxiv.org/abs/1710.09412`. Claimed to be possible due to adaptive gradient clipping

  Mixup regularization idea: use interpolated pairs of samples $(\tilde{x}, \tilde{y})$. They are drawn from randomly drawn pairs of labeled training samples: (x_i, y_i), x_k, y_k by standard linear interpolation:

$$\lambda \sim Beta(\alpha, \alpha)$$
$$\tilde{x} = \lambda x_i + (1 - \lambda) x_k$$
$$\tilde{y} = \lambda y_i + (1 - \lambda) y_k$$

- section C appendix: many small changes to the architecture of SE-Resnext (no need to memorize those). For example, changed channel sizes in the resnext layer1 to layer4: before $256, 512, 1024, 2048$, new: $256, 512, 1536, 1536$