IN5400 – Optimizers and Data Augmentation

Alexander Binder

February 23, 2021

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 Efficient Net components

- depth-wise separable convolution: be able to explain the difference in model design between depth-wise separable convolution with 1×1 convolution and conventional convolution
- depth-wise separable convolution: understand the difference in terms of computation efforts and parameter counts between a block of depth-wise separable convolution followed by a 1 convolution versus a single conventional convolution
- squeeze-and-excitation module as plugin

Lets see if we do this in this or in the next lecture:

teacher student learning and noisy students

- how to use noisy student approach
- be able to explain the basic steps how teacher student training works

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

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

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

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`

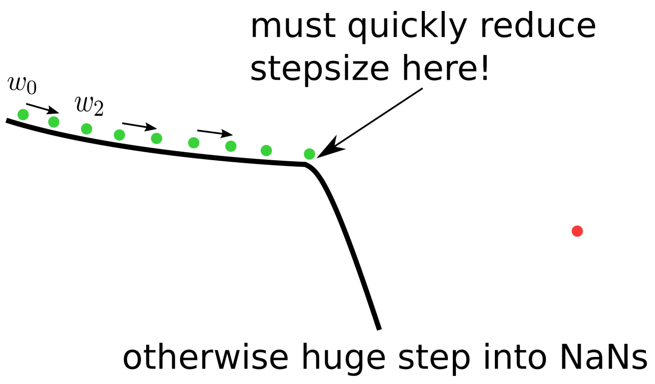
- DIVERGENCE if learning rate too high - (see example in past lecture)
- in a flat region steps can be very small:

  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


must quickly reduce stepsize here!
w_0
w_2
otherwise huge step into NaNs

- 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)

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

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

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

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 $\eta^{(i)}_{max}$ 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`

1.3 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.

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

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:

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

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

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

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

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 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.

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

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 $1-\alpha^{t+1}$.

1.6 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:

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

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.

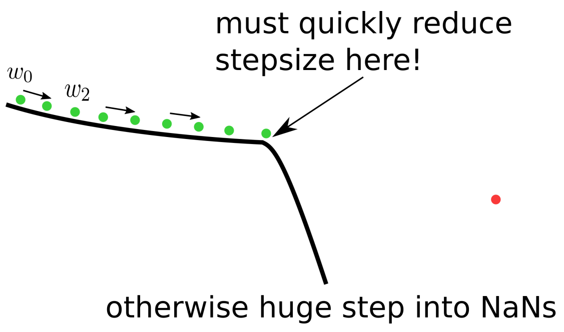
So use an average of norms of gradients from the past, and divide by them. Divide by $\text{EMA}(\cdot)_t$ of norms of gradients:

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

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.



must quickly reduce stepsize here!
w_0
w_2
otherwise huge step into NaNs

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 ?

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

Compate to RMSprop:

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

it does something like:

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

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

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

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

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

1.7 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?

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

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 $g_t := \nabla_w \hat{E}(w_t, L)$
    $s_t[d] = \alpha_1 s_{t-1} + (1-\alpha_1)(g_t[d])^2$ #EMA of $g_t[d]^2$
    $s_t = (s_t[0], \ldots, s_t[D]) = (s_t[d])_d$ #vectorization
    $m_t = \alpha_2 m_{t-1} + (1-\alpha_2) g_t$ #EMA of $g_t[d]$vectorized
    $w_{t+1} = w_t - \eta_t \frac{m_t}{\sqrt{s_t} + \epsilon}$
        $1/\sqrt{s_t}$ : (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 $\eta, \epsilon, \alpha_1, \alpha_2$

$$
\begin{aligned}
m_0 &= 0, s_0 = 0 \\
\text{compute } g_t &:= \nabla_w \hat{E}(w_t, L) \\
s_t[d] &= \alpha_1 s_{t-1}[d] + (1-\alpha_1)(g_t[d])^2 \quad \#element-wise \\
m_t &= \alpha_2 m_{t-1} + (1-\alpha_2) g_t \\
c_{t,1} &= 1-\alpha_1^t, c_{t,2} = 1-\alpha_2^t \\
w_{t+1} &= w_t - \eta_t \frac{m_t/c_{t,1}}{\sqrt{s_t/c_{t,2}}+\epsilon} \\
1/\sqrt{s_t} &: \ \text{(element-wise division for every dimension } s_t[d])
\end{aligned}
$$

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 $+\epsilon$.

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

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

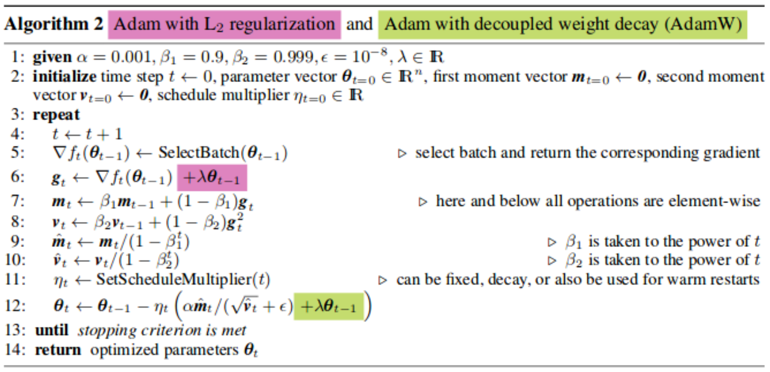
$$m_t = \alpha_2 m_{t-1} + (1-\alpha_2) g_t$$

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

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

$1/\sqrt{s_t}$: (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)
 1: given $\alpha = 0.001, \beta_1 = 0.9, \beta_2 = 0.999, \epsilon = 10^{-8}, \lambda \in \mathbb{R}$
 2: 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 $\eta_{t=0} \in \mathbb{R}$
 3: repeat
 4:     $t \leftarrow t + 1$
 5:     $\nabla f_t(\boldsymbol{\theta}_{t-1}) \leftarrow$ SelectBatch($\boldsymbol{\theta}_{t-1}$)      ▷ select batch and return the corresponding gradient
 6:     $\boldsymbol{g}_t \leftarrow \nabla f_t(\boldsymbol{\theta}_{t-1})$ $+\lambda\boldsymbol{\theta}_{t-1}$
 7:     $\boldsymbol{m}_t \leftarrow \beta_1 \boldsymbol{m}_{t-1} + (1-\beta_1)\boldsymbol{g}_t$      ▷ here and below all operations are element-wise
 8:     $\boldsymbol{v}_t \leftarrow \beta_2 \boldsymbol{v}_{t-1} + (1-\beta_2)\boldsymbol{g}_t^2$
 9:     $\hat{\boldsymbol{m}}_t \leftarrow \boldsymbol{m}_t/(1-\beta_1^t)$      ▷ $\beta_1$ is taken to the power of $t$
10:     $\hat{\boldsymbol{v}}_t \leftarrow \boldsymbol{v}_t/(1-\beta_2^t)$      ▷ $\beta_2$ is taken to the power of $t$
11:     $\eta_t \leftarrow$ SetScheduleMultiplier($t$)      ▷ can be fixed, decay, or also be used for warm restarts
12:     $\boldsymbol{\theta}_t \leftarrow \boldsymbol{\theta}_{t-1} - \eta_t \left(\alpha \hat{\boldsymbol{m}}_t/(\sqrt{\hat{\boldsymbol{v}}_t} + \epsilon) +\lambda\boldsymbol{\theta}_{t-1}\right)$
13: until stopping criterion is met
14: 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.

1.9 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

1.10 Where are those in pytorch?

`torch.optim`

1.11 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

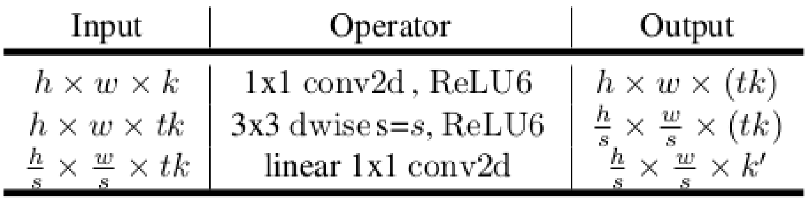
today:

- blocks of depth-wise separable convolution with 1×1 convolution
- squeeze-and-excitation module as plugin
- Efficientnet `https://arxiv.org/abs/1905.11946`

... Approach:

- take a lightweight neural network building block (MobileNet inverted residual)
- search for an initial neural architecture *EfficientNet-B0* based on it
- then expand the neural network by at the same time: –increasing depth (more layers), –increasing width (more channels in each layer) and – increasing spatial resolution
- train the upscaled networks

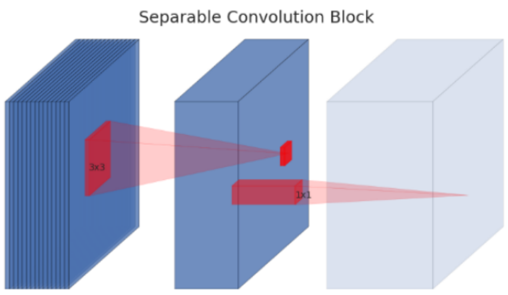
2.1 Efficient Net – the Basic Block: MobileNet inverted residual block



Input	Operator	Output
$h \times w \times k$	1x1 conv2d , ReLU6	$h \times w \times (tk)$
$h \times w \times tk$	3x3 dwise s=s, ReLU6	$\frac{h}{s} \times \frac{w}{s} \times (tk)$
$\frac{h}{s} \times \frac{w}{s} \times tk$	linear 1x1 conv2d	$\frac{h}{s} \times \frac{w}{s} \times k'$

Table 1: *Bottleneck residual block* transforming from k to k' channels, with stride s, and expansion factor t.

The inspiration of this stacking of conv2d(1×1)–ReLU–conv2d(depth-wise separable $k \times k$)–ReLU–conv2d(1×1) are **depth-wise separable convolutions**

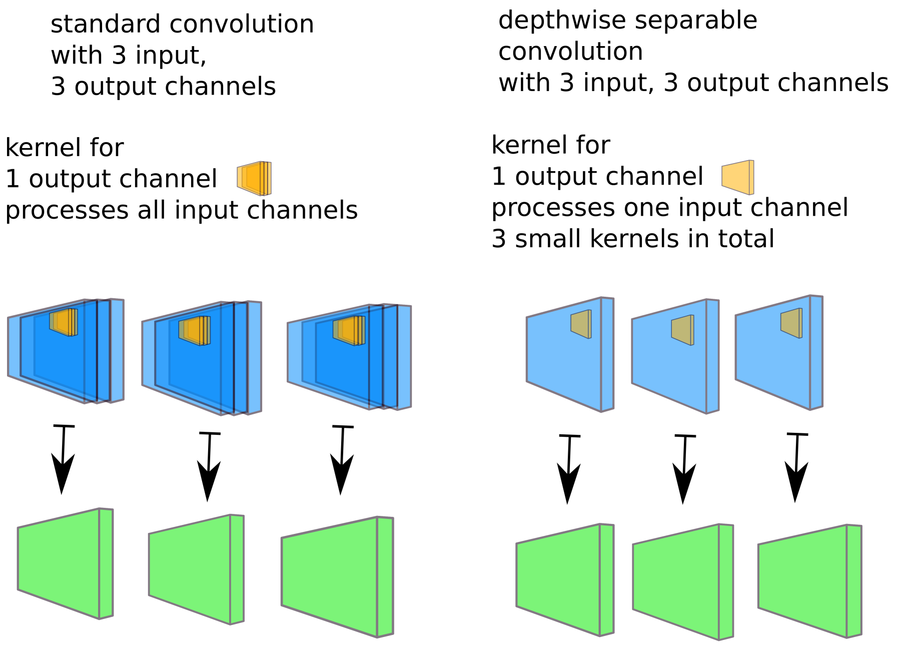


Left: depthwise separable convolution. Middle: 1×1-convolution

Processing intuition:

- The depthwise separable convolution processes each input channel separately in its spatial neighbors
- the 1×1-convolution allows channels to interact with each other.

This is a replacement for a 3×3 convolution. To see why, let us compare standard convolution vs the block of depthwise separable $+1 \times 1$ convolutions:



- depthwise separable convolution: applies a kernel k_c with one input and output channel of size $(c_{in} = 1, size_h = k, size_w = k, c_{out} = 1)$ separately to each input channel of the input feature map. The kernel k_c is different for each input channel index c. Number of parameters: $c * k^2$
- computational cost for a concatenation of conv2d(depth-wise separable $k \times k) - 1 \times 1$ convolution is much less than a full convolution:
  - assume: the whole blockmaps (c, h, w) to (c', h, w) with kernel size k, that is, the conv2d(1×1) maps c to c'.
  - computational cost for a full convolution: $c * h * w * k^2 * c'$.

    Why ? $k \times k$ products over $h * w$ positions (approximately for stride 1, depends on padding, we assume canonical padding). this for c input channels and c' output channels
  - computational cost for a block of d-wise separable $k \times k + 1 \times 1$-convolution: $chwk^2 + chwc' = chw(k^2 + c')$ – one order less.

    Why ?

depth-wise separable convolution is applying hwk^2 multiplications for c kernels with 1 input channel for each kernel: results in $chwk^2$

The conv2d(1×1) needs $chwc'$ multiplications over a feature map of size (h, w) and c input and c' output channels.

- trainable parameters?
  - full convolution $c \cdot k^2 \cdot c'$
  - a block of d-wise separable $k \times k$ + 1×1-convolution: $c \cdot k^2$ (c kernels of size k, one for each input channels) $+c * c'$ $(1 \times 1$ conv)

  total: $c(k^2 + c')$ – same one order less
- modified architecture with a bottleneck:
  - depth-wise convolution with kernel $(1, k, k, 1)$ - ReLU -
  - 1×1 convolution (to a small output channel number = the bottleneck) - ReLU -
  - 1×1 convolution (to a larger output channel number = the expansion)

(c) Separable with linear bottleneck

(d) Bottleneck with expansion layer

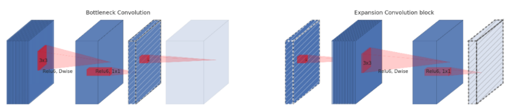
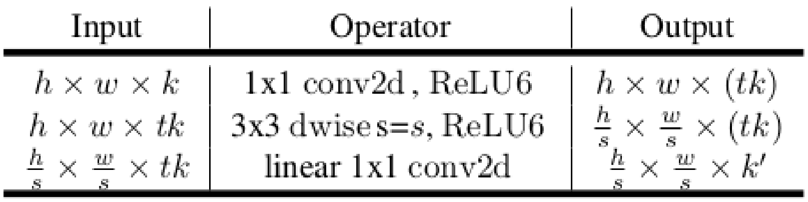


Figure 2: Evolution of separable convolution blocks. The diagonally hatched texture indicates layers that do not contain non-linearities. The last (lightly colored) layer indicates the beginning of the next block. Note: 2d and 2c are equivalent blocks when stacked. Best viewed in color.

Left: depthwise separable convolution, followed by a 1×1-convolution as a bottleneck, followed by a 1×1-convolution to expand to a larger channel size

- modified architecture with bottleneck - stack 3
  - depth-wise convolution with kernel $(1, k, k, 1)$ - ReLU -
  - 1×1 convolution (to a small output channel number = the bottleneck) - ReLU -
  - 1×1 convolution (to a larger output channel number = the expansion)
  - depth-wise convolution with kernel $(1, k, k, 1)$ - ReLU -
  - 1×1 convolution (to a small output channel number = the bottleneck) - ReLU -
  - 1×1 convolution (to a larger output channel number = the expansion)
  - depth-wise convolution with kernel $(1, k, k, 1)$ - ReLU -
  - 1×1 convolution (to a small output channel number = the bottleneck) - ReLU -
  - 1×1 convolution (to a larger output channel number = the expansion)

- modified architecture with bottleneck - stack 3, now start from the third item:
  - 1×1 convolution (to a larger output channel number = the expansion)
  - depth-wise convolution with kernel $(1, k, k, 1)$ - ReLU -
  - 1×1 convolution (to a small output channel number = the bottleneck) - ReLU -
  - 1×1 convolution (to a larger output channel number = the expansion)
  - depth-wise convolution with kernel $(1, k, k, 1)$ - ReLU -
  - 1×1 convolution (to a small output channel number = the bottleneck) - ReLU -
  - add a shortcut from start to end $x + H(x)$
  - Why this mod? If computation passes through a bottleneck, then all the information must be contained in the bottleneck already, therefore we can start at the bottleneck, having a smaller feature map, rather than starting at the larger feature map – mobileNet ;)
- MobileNet inverted residual block:



Input	Operator	Output
$h \times w \times k$	1x1 conv2d , ReLU6	$h \times w \times (tk)$
$h \times w \times tk$	3x3 dwise s=s, ReLU6	$\frac{h}{s} \times \frac{w}{s} \times (tk)$
$\frac{h}{s} \times \frac{w}{s} \times tk$	linear 1x1 conv2d	$\frac{h}{s} \times \frac{w}{s} \times k'$

Table 1: *Bottleneck residual block* transforming from k to k' channels, with stride s, and expansion factor t.

similar idea to depth-wise convolution - ReLU - 1×1 convolution - ReLU.
inverted residual:

- 1×1 2d conv expands number of channels
- $k \times k$ 2d depthwise-separable conv processes each channel with its own filter acting only on a single input channel
- 1×1 2d conv shrinks back number of channels
- add a residual-making shortcut from start to end $x + H(x)$
- same idea: low computations and ...
- the bottleneck feature maps with small number of channels contain the actual information
- before summing residual added a squeeze-and-excitation block

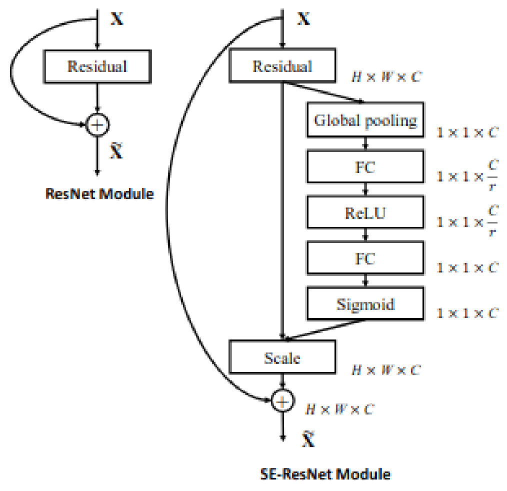


Fig. 3. The schema of the original Residual module (left) and the SE-ResNet module (right).

idea behind it:

- compute a reweighting of each channel s_c

$$\begin{aligned} z_c &= \text{Pool}(f[c, :, :]) \\ s &= \sigma(W_2 \text{ReLU}(W_1 z)) \end{aligned}$$

- use 1 feature per channel (z_c) as input
- s learns a reweighting by letting channels interact with each other
- multiply output of residual block with the reweighting:

$$\tilde{u} = s_c u_c$$

- then add residual on top of that
- light-weight module, s weights do not sum to 1, component-wise rescaling rather than attention weights
- note the bottleneck C/r

2.2 EfficientNet B0

found by a neural architecture search from MobileNet inverted residual blocks with different strides and kernel sizes

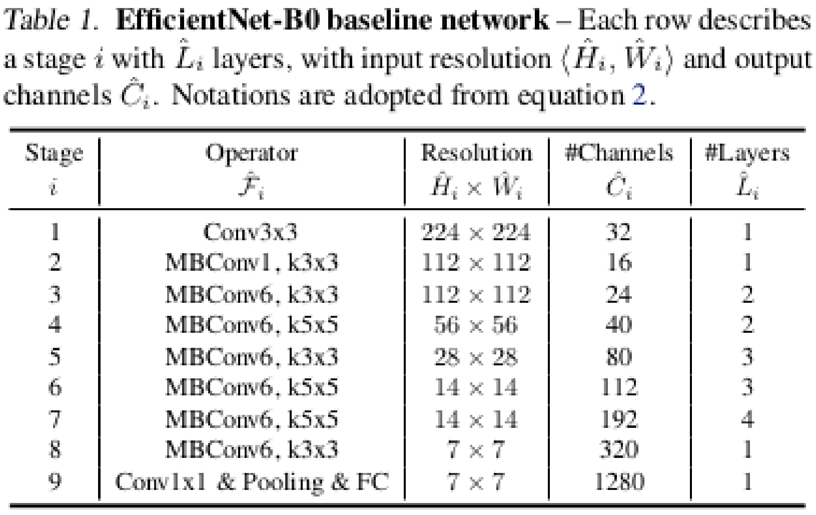


Table 1. **EfficientNet-B0 baseline network** – Each row describes a stage i with $\hat{L}_i$ layers, with input resolution $\langle \hat{H}_i, \hat{W}_i \rangle$ and output channels $\hat{C}_i$. Notations are adopted from equation 2.

Stage i	Operator $\hat{\mathcal{F}}_i$	Resolution $\hat{H}_i \times \hat{W}_i$	#Channels $\hat{C}_i$	#Layers $\hat{L}_i$
1	Conv3x3	224×224	32	1
2	MBConv1, k3x3	112×112	16	1
3	MBConv6, k3x3	112×112	24	2
4	MBConv6, k5x5	56×56	40	2
5	MBConv6, k3x3	28×28	80	3
6	MBConv6, k5x5	14×14	112	3
7	MBConv6, k5x5	14×14	192	4
8	MBConv6, k3x3	7×7	320	1
9	Conv1x1 & Pooling & FC	7×7	1280	1

2.3 Efficient Net Bi, $i \geq 1$ via expansion

Let be $\phi \in \{1, \ldots, 7\}$ the expansion coefficient. Scale neural network depth (number of layers), neural network width (number of channels in each feature map), and spatial resolution of all feature maps by multiplied factors of:

$$\begin{aligned}
\text{depth: } & d = \alpha^{\phi} \\
\text{width: } & w = \beta^{\phi} \\
\text{resolution: } & r = \gamma^{\phi} \\
\alpha, \geq 1,\ \beta \geq 1,\ & \gamma \geq 1, \\
& \alpha\beta^2\gamma^2 \approx 2
\end{aligned}$$

- STEP 1:first fix $\phi = 1$, assuming twice more re-sources available, and do a small grid search of α, β, γ. In particular, they found the best values for EfficientNet-B0 are $\alpha = 1.2, \beta = 1.1, \gamma = 1.15$, under constraint of $\alpha\beta^2\gamma^2 \approx 2$
- STEP 2: then fix α, β, γ as constants and scale up baseline network with different ϕ, to obtain EfficientNet-B1 to B7

Why $\alpha\beta^2\gamma^2 \approx 2$?

- if using $\phi = 1$, and upscaling the network by α, β, γ then the computational costs increased by $\alpha\beta^2\gamma^2$ and we want the computational cost to double in one step.
- γ^2 is obvious because we have 2 spatial dimensions to which γ is applied.
- β^2 is because the 1×1 conv complexity of computations is $hwcc'$ – this scales quadratically if $c = c'$

How to get to EfficientNet Bi

- for every i train the EfficientNet Bi from scratch.
- Results ? See paper.
- (off exams) See in the paper "5.2 ImageNet Results for EfficientNet" for many additional tricks

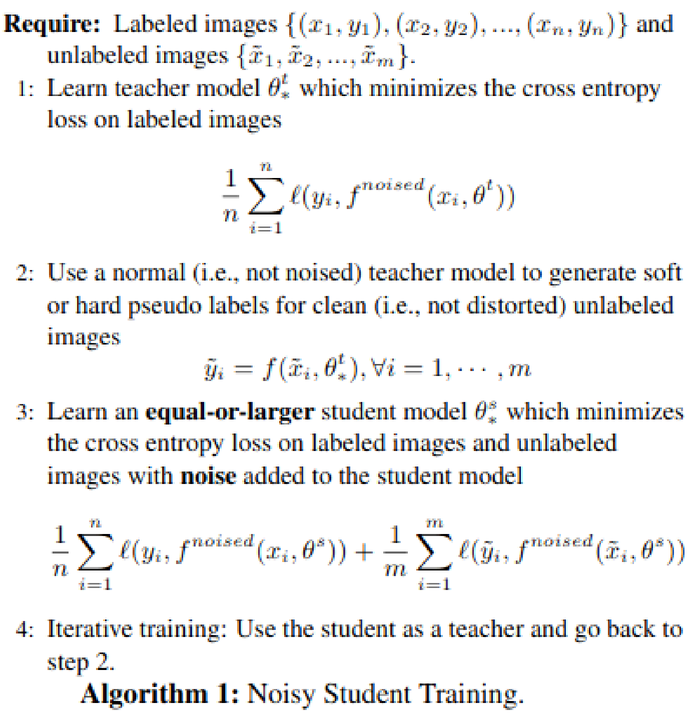
One final remark (off exams):

convolution layers, stacking simple elements deeply, residuals – follows ideas to enforce a structured, hierarchical learning. Will see this in GANs again.

In theory a dense layer could represent a convolution operation (by having tied weights, which are only zero around a center).
Gradient flow improvements (batchnorm, residuals), optimizers, teacher student learning – also about making learning more uniform / normalized, hopefully easier.
Compare later to RNNs: CNNs: gradient flow to be preserved through many layers. RNNs: gradient flow to be preserved through many time steps – where the hidden state at a time t can be compared to a CNN feature map at layer index t :)

2.4 Noisy student

Xie et al. `https://arxiv.org/abs/1911.04252`



```
Require: Labeled images $\{(x_1, y_1), (x_2, y_2), ..., (x_n, y_n)\}$ and
    unlabeled images $\{\tilde{x}_1, \tilde{x}_2, ..., \tilde{x}_m\}$.
1: Learn teacher model $\theta^t_*$ which minimizes the cross entropy
   loss on labeled images
       $\frac{1}{n}\sum_{i=1}^{n} \ell(y_i, f^{noised}(x_i, \theta^t))$
2: Use a normal (i.e., not noised) teacher model to generate soft
   or hard pseudo labels for clean (i.e., not distorted) unlabeled
   images
       $\tilde{y}_i = f(\tilde{x}_i, \theta^t_*), \forall i = 1, \cdots, m$
3: Learn an **equal-or-larger** student model $\theta^s_*$ which minimizes
   the cross entropy loss on labeled images and unlabeled
   images with **noise** added to the student model
       $\frac{1}{n}\sum_{i=1}^{n} \ell(y_i, f^{noised}(x_i, \theta^s)) + \frac{1}{m}\sum_{i=1}^{m} \ell(\tilde{y}_i, f^{noised}(\tilde{x}_i, \theta^s))$
4: Iterative training: Use the student as a teacher and go back to
   step 2.
```

Algorithm 1: Noisy Student Training.

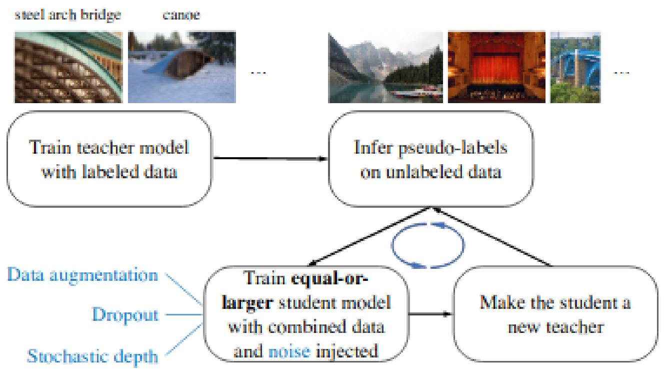

steel arch bridge
canoe
...
...
Train teacher model with labeled data
Infer pseudo-labels on unlabeled data
Data augmentation
Dropout
Stochastic depth
Train equal-or-larger student model with combined data and noise injected
Make the student a new teacher

Figure 1: Illustration of the Noisy Student Training. (All shown images are from ImageNet.)

nothing more to say

- makes use of additional unlabeled data, labels it by the teacher to obtain probability distribution per image as label
- uses soft labels computed by the teacher on original images, but for training adds noise to the student model for the same images

- combines semi-supervised data, weakly supervision and model noising
- Personal guess: probably needs a sufficient strong teacher to start with. likely wont work with poorly performing teachers...

2.5 Background: teacher-student training

- goal: want to train a network (the student)
- idea: do not train it using hard ground truth labels. Given a sample x, obtain softmax probability distribution $t(x)$ from the teacher first
- train student $s(\cdot)$ with the teacher probabilities $t(x)$ as soft labels for a loss capable to do so, e.g. using cross-entropy

  Reminder: Cross entropy of two distributions p, s:

  $$CE(p,q) = \sum_{c=1}^{C} -p_c \log s_c$$

  Reminder: if p are one-hot ground truth labels, and $s = s(x)$ is the softmax prediction over an image, this reduces to:

  $$-\log s_c(x) \text{ where } c \text{ is the ground truth class of } x$$

  the neg log probability of the prediction for the ground truth class.

  The cross-entropy between teacher and student are the teacher-weighted neg log student probabilities:

  $$L(x, t(x)) = \sum_{c=1}^{C} -t_c(x) \log s_c(x)$$

- why does it work well ? effectively prioritizes the learning of easy samples! (Out of class: see curriculum learning in machine learning). The losses are lower for hard samples where $t_c(x)$ is close to the guessing threshold $\frac{1}{C}$ for most classes. Losses are larger for easy samples, where the teacher is very confident and predicts $t(x)$ close to a one-hot-label.

  Idea here: prioritize the learning of easy samples. (out of class, opposite idea: hard negative mining for problems with too many negative samples)

- applications: usually used to train structurally smaller/lighter students from complex teacher models, e.g. to obtain a faster or more resource efficient model (Noisy student trains a model of the same complexity)