Training DNN: from theory to practice

Gabriel Synnaeve, **Alexandre Défossez**

March 11th 2022

LSML @ Mines ParisTech

facebook
Artificial Intelligence Research

Outline

1. Non convex stochastic optimization
2. Distributed optimization
3. Adaptive optimization
4. Regularization of DNN
5. Practical DNN training

Non Convexe Stochastic Optimization

SGD for deep neural networks

SGD for non convex optimization

Notations

Let $F(x) = \mathbb{E}[\mathbf{f}(x)]$ for $\mathbf{f} : \mathbb{R}^d \mapsto \mathbb{R}$ a random variable.

$x \in \mathbb{R}^d$ are the **weights** of the model.

$\mathbf{f}$ is the loss over 1 **training example** at random.

F is the loss over the entire training set.

SGD for non convex optimization

Algorithm

For $x_0 \in \mathbb{R}^d$, we define iteratively for all iteration $n \in \mathbb{N}$,

$$x_{n+1} = x_n - \gamma \nabla f_n(x_n),$$

for a step size $\gamma > 0$, and taking $f_n \sim \mathbf{f}$ i.i.d.

SGD for non convex optimization

Assumptions

We make the following assumptions:

1. F is **lower bounded** by F_*
$$\forall x \in \mathbb{R}^d, F(x) \geq F_*$$

2. F is L**-smooth**, i.e. ∇F is L-Liptchiz
$$\forall x, y \in \mathbb{R}^d, \|\nabla F(x) - \nabla F(y)\| \leq L\|x - y\|$$

3. The **variance** of gradient is **bounded** by σ^2
$$\forall x \in \mathbb{R}^d, \mathbb{E}\left[\|\nabla \mathbf{f}(x) - \nabla F(x)\|^2\right] \leq \sigma^2$$

SGD for non convex optimization

Convergence

[Ghadimi et Lan, 2013]

Let $\tau \sim \mathcal{U}(0,\ldots,N-1)$ a random time **uniformly** distributed over $\{0,\ldots,N-1\}$.

Under the 3 assumptions stated before and if $\gamma < L$, we have

$$\mathbb{E}\left[\|\nabla F(x_\tau)\|^2\right] \leq 2\frac{F(x_0) - F_*}{\gamma N} + \gamma L \sigma^2,$$

In particular, taking $\gamma = 1/\sqrt{N}$ (for N sufficiently large), we get

$$\mathbb{E}\left[\|\nabla F(x_\tau)\|^2\right] \leq 2\frac{F(x_0) - F_*}{\sqrt{N}} + \frac{L\sigma^2}{\sqrt{N}} = O\left(\frac{1}{\sqrt{N}}\right)$$

Proof (for the curious)

Using the **smoothness** of F, we have

$$F(x_{n+1}) \leq F(x_n) - \gamma \nabla f_n(x_n)^T \nabla F(x_n) + \frac{1}{2}\gamma^2 L \|\nabla f_n(x_n)\|^2.$$

Smoothness formula:

$$F(y) = F(x) + \int_x^y \nabla F(z)^T \mathrm{d}z\,.$$

Defined as integral for $t \in [0,1]$ with $z = ty + (1-t)x$, $\mathrm{d}z = (y-x)\mathrm{d}t$.

$\nabla F(z) = \nabla F(x) + \int_x^z \nabla^2 F(u)\mathrm{d}u$. Gradient is Liptchiz $\leftrightarrow$ $\nabla^2 F \preceq L$

$\|\nabla F(z) - \nabla F(x)\| \leq L\|z - x\|$. Replace z with expression above, Inject into $F(y)$ formula, and you get the result.

Taking the expectation **conditionally** on $(f_1, \ldots, f_n)$ (noted $\mathbb{E}_n$), we get

$$\mathbb{E}_n[F(x_{n+1})] \leq F(x_n) - \gamma \nabla F(x_n)^T \nabla F(x_n) + \frac{1}{2}\gamma^2 L \|\nabla F(x_n)\|^2 + \frac{1}{2}\gamma^2 L \sigma^2,$$

Moving around the term

$$\gamma \|\nabla F(x_n)\|^2 \left(1 - \frac{\gamma L}{2}\right) \leq F(x_n) - \mathbb{E}_n[F(x_{n+1})] + \frac{1}{2}\gamma^2 L \sigma^2.$$

Now we sum over all $n \in \{0, \ldots, N-1\}$, and take the full expectation. Note the blue terms **telescopes** !

$$\sum_{n=0}^{N-1} \gamma \mathbb{E}\left[\|\nabla F(x_n)\|^2\right] \left(1 - \frac{\gamma L}{2}\right) \leq F(x_0) - F_* + \frac{1}{2} N \gamma^2 L \sigma^2.$$

(We used $F(x_N) \geq F_*$). Then using the condition on the step size γ and minor rewrites gives us the result !

SGD for non convex optimization

Regimes

$$\mathbb{E}\left[\|\nabla F(x_\tau)\|^2\right] \leq 2\frac{F(x_0) - F_*}{\gamma N} + \gamma L \sigma^2,$$

Forgetting of the initial condition vs. asymptotic random walk.

For x_0 far from optimum, first term dominates (early training).

Large step size: moves away from x_0 faster !

If we initialize to x_* s.t. $F(x_*) = F_*$, optimal step size: $\gamma = 0$.

If $\gamma > 0$, **random walk** around x_* (as gradient is 0 on average).

SGD for non convex optimization

Regimes

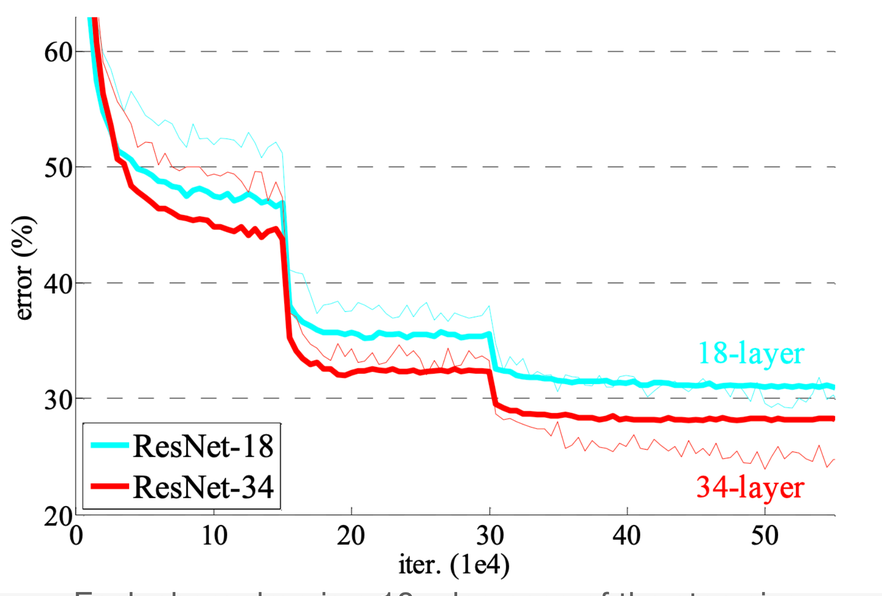
$$\mathbb{E}\left[\|\nabla F(x_\tau)\|^2\right] \leq 2\frac{F(x_0) - F_*}{\gamma N} + \gamma L \sigma^2,$$

Forgetting of the initial condition vs. asymptotic random walk.

For DNN training, first regime is most important.

Large constant step size used first.

Theoretically, decrease step size only if training loss stops improving. In practice, if we use valid loss because of **overfitting**.



Each sharp drop is a 10x decrease of the step size.
Bold is validation and thin is training.
Credit: [He et al. 2015].

Distributed optimization

What theory tells us

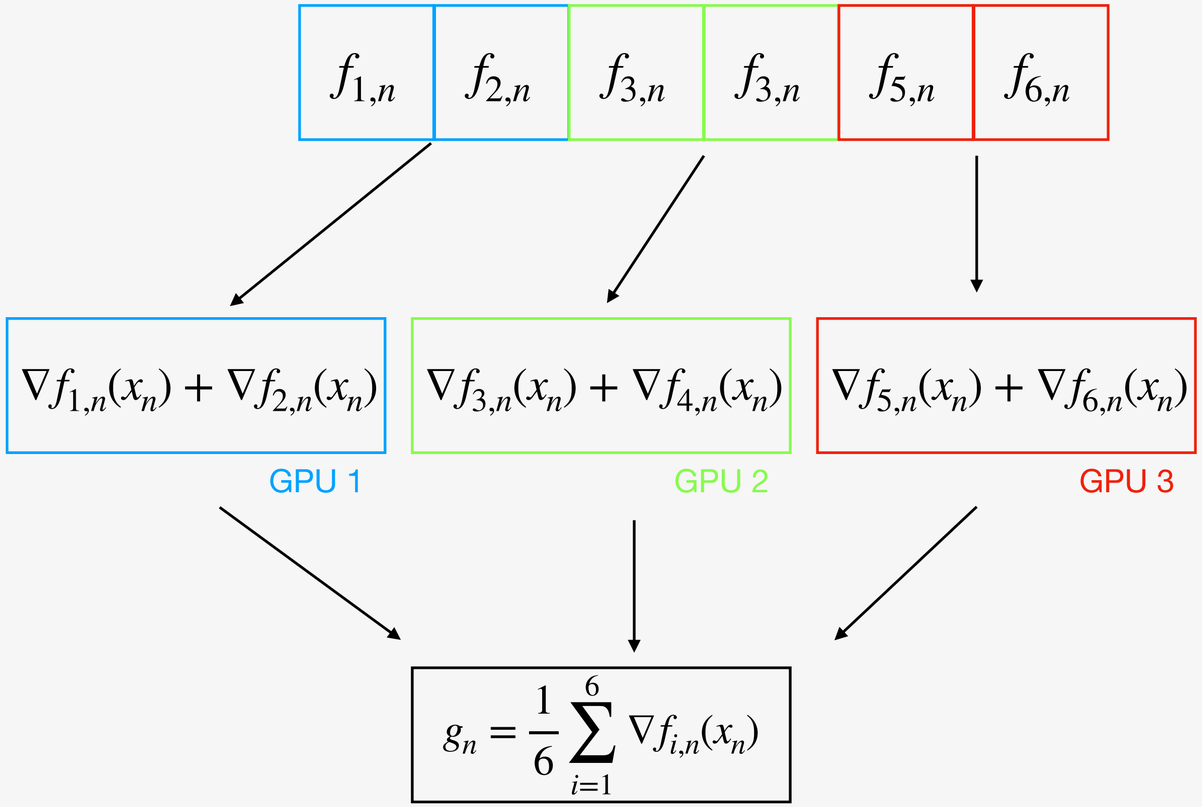
Distributed training for DNN

Synchronous distributed SGD

Instead of sampling single f_n from training set, sample **batch** of size B: $f_{1,n}, f_{2,n}, , \ldots, f_{B,n}, .$

Given W process, each with a gpu, **dispatch** B/W over each machine.

Average gradient across machines, update model and restart.



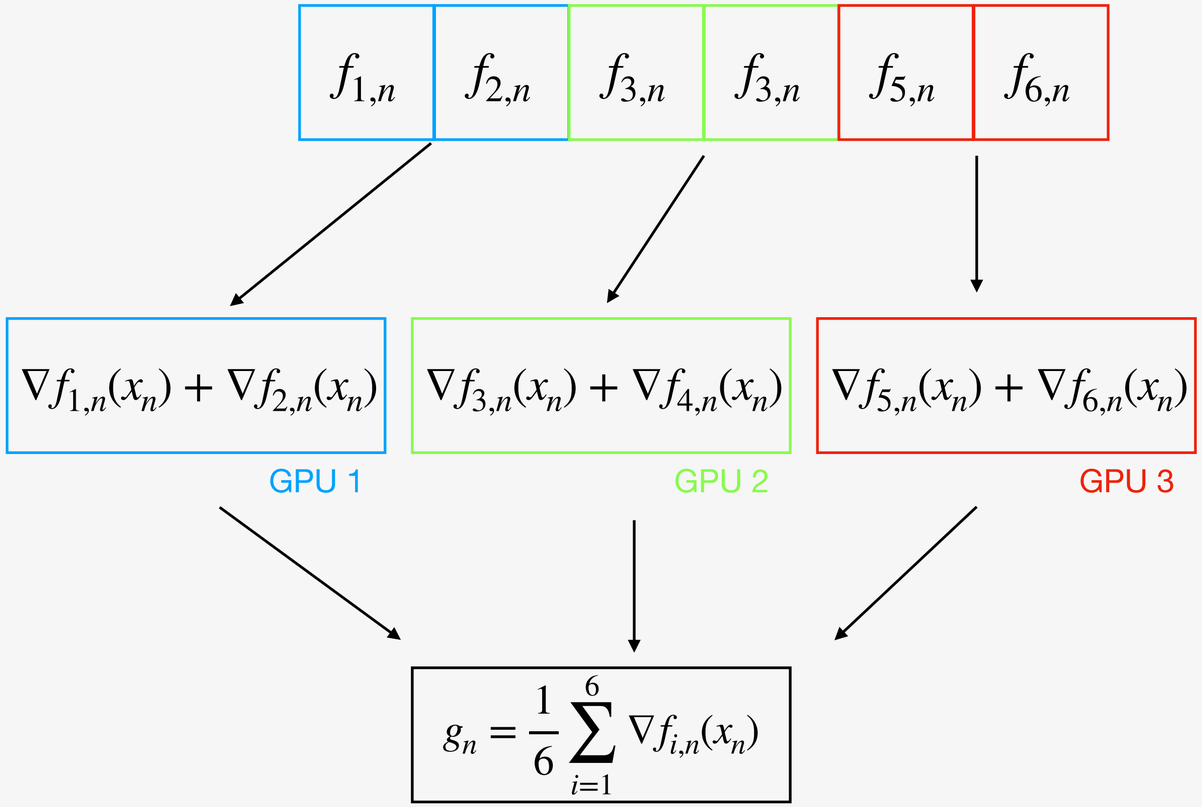
Distributed training for DNN

Synchronous distributed SGD

Advantage: simple, **same theory** as single GPU.

Disavantage: need to **wait** on g_n to be fully computed and averaged before starting g_{n+1}.

Idea: **asynchronous** updates ? Theory is complex and in practice doesn't work better !



Speedup of Synchronous SGD

The impact of mini-batching

N is total number of **iterations**, not samples !

Variance for batch size B is reduced $\sigma_B^2 = \dfrac{\sigma^2}{B}$.

We note $T_{B,W}$ **process time** for a batch of size B with W workers.

Ideally $T_{WB,W} = T_{B,1}$.

In practice, $T_{WB,W} \leq T_{B,1}$ due to communication **latency**.

$$\mathbb{E}\left[\|\nabla F(x_\tau)\|^2\right] \leq 2\frac{F(x_0) - F_*}{\gamma N} + \gamma L \sigma^2$$

For batch size B:

$$\mathbb{E}\left[\|\nabla F(x_\tau)\|^2\right] \leq 2\frac{F(x_0) - F_*}{\gamma N} + \frac{\gamma L \sigma^2}{B}$$

Speedup of Synchronous SGD

The iteration vs. variance trade-off

If W workers, taking $\tilde{B} = WB$, we have

$T_{\tilde{B},W} = T_{B,1}$. Given total **time budget** T, we can process W more samples, but nb of iterations it still the same.

$$\mathbb{E}\left[\|\nabla F(x_\tau)\|^2\right] \leq 2\frac{F(x_0) - F_*}{\gamma N} + \frac{\gamma L \sigma^2}{BW}$$

For W large, variance term is ≈ 0, no more gains.

No magic: gain only up to a point, then **plateau** !

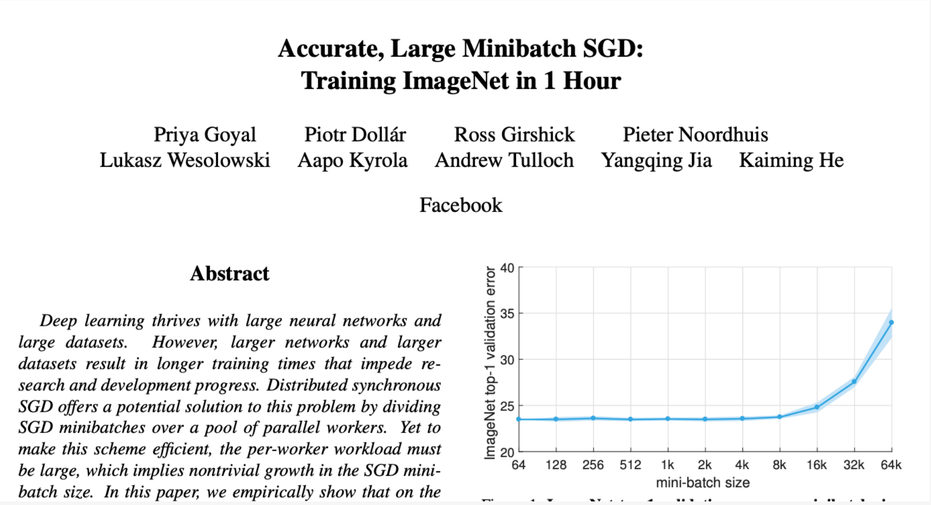
Extreme speed-up in practice

Due to high variance, initial batch size ~ 64.

Up to 4 to 8 GPUs (depends on model complexity) distribution keeping same batch size requires no change.

Beyond 8 GPU, use following tricks:

- Increase batch size by factor K.
- Increase learning rate by factor K or $\sqrt{K}$ if diverges.
- Gradual warmup of learning rate.



Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour

Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, Kaiming He

Facebook

Abstract

Deep learning thrives with large neural networks and large datasets. However, larger networks and larger datasets result in longer training times that impede research and development progress. Distributed synchronous SGD offers a potential solution to this problem by dividing SGD minibatches over a pool of parallel workers. Yet to make this scheme efficient, the per-worker workload must be large, which implies nontrivial growth in the SGD mini-batch size. In this paper, we empirically show that on the

An example of extreme speed up on image net, and the tricks required.
Credit: [Goyal et al. 2018].

Adaptive optimization

A single learning rate to rule them all !

The parametrization issue

How to mess up with SGD.

Let $F(x) = \mathbb{E}[\mathbf{f}(x)]$ for $\mathbf{f} : \mathbb{R}^d \mapsto \mathbb{R}$ a random variable.

Let us take $\lambda \in \mathbb{R}^*$. We define $G(y) = F(\lambda y)$, and $\mathbf{g}(\mathbf{y}) = \mathbf{f}(\lambda y)$.

This is a **scalar reparametrization** of the original function space.

$$\nabla \mathbf{g}(y) = \lambda \nabla f(\lambda y)$$

The parametrization issue

How to mess up with SGD.

$G(y) = F(\lambda y)$, and $\mathbf{g}(\mathbf{y}) = \mathbf{f}(\lambda y)$. We define $\tilde{x} = \lambda y$.

$$\nabla \mathbf{g}(y) = \lambda \nabla f(\lambda y)$$

$$G(y + \gamma \nabla \mathbf{g}(y)) = F(\lambda(y + \gamma \lambda \nabla \mathbf{f}(\lambda y)))$$

$$G(y + \gamma \nabla \mathbf{g}(y)) = F(\tilde{x} + \gamma \lambda^2 \nabla \mathbf{f}(\tilde{x}))$$

Intuitively: λ factor from gradient (backward), and another in the forward !

SGD over $G(y)$ is equivalent to SGD over $F(\tilde{x})$ with step size $\gamma \lambda^2$.

The parametrization issue

How to mess up with SGD.

$G(y) = F(\lambda y)$, and $\mathbf{g}(\mathbf{y}) = \mathbf{f}(\lambda y)$. We define $\tilde{x} = \lambda y$.

SGD over $G(y)$ is equivalent to SGD over $F(\tilde{x})$ with step size $\gamma\lambda^2$.

If $\lambda \ll 1$: no learning. If $\lambda \gg 1$, divergence !

Ideal optimization: result independant of λ (second order, Newton method, natural gradient etc.).

But: doesn't work for non convex, doesn't work for stochastic :'(

Adaptive methods

A partial solution to the parametrization issue

Adagrad [Duchi et al. 2011]:

$$\begin{cases} x_{n+1} &= x_n - \gamma \frac{\nabla f_n(x_n)}{\sqrt{\epsilon + v_{n+1}}} \\ v_{n+1} &= v_n + \left(\nabla f_n(x_n) \right)^2 \end{cases}$$

The division and squaring are **per dimension** !

One **effective step size** $\gamma \sqrt{\epsilon + v_n^2}^{-1}$ per dimension.

Converges as $O\left(1/\sqrt{N}\right)$ for any γ. No need to know L.

Adaptive methods

A partial solution to the parametrization issue

Taking back $g_n(y) = f_n(\lambda y)$. Let us denote $\tilde{x} = \lambda^{-1} y$

$$\begin{cases} y_{n+1} & = y_n - \gamma \dfrac{\lambda \nabla f_n(\tilde{x})}{\sqrt{\epsilon + v_{n+1}}} \\ v_{n+1} & = v_n + \lambda^2 \left(\nabla f_n(\tilde{x}) \right)^2 \end{cases}$$

v_n is scaled by λ^2, which cancels the numerator !

λ factor is canceled in the backward, only impacts in the foward.

No **amplification** as λ^2 as with SGD.

Adam

Adaptive optimization for DNN

Effective learning rate decreases quickly with Adagrad, because v_n is always increasing. Instead Adam uses **exponential moving average**:

$$\begin{cases} x_{n+1} & = x_n - \gamma \dfrac{\tilde{m}_{n+1}}{\sqrt{\epsilon + \tilde{v}_{n+1}}} \\ m_{n+1} & = \beta_1 m_n + (1-\beta_1)\nabla f_n(x_n) \\ v_{n+1} & = \beta_2 v_n + (1-\beta_2)\left(\nabla f_n(x_n)\right)^2 \end{cases} \quad \text{with} \quad \begin{cases} \tilde{m}_n & = \dfrac{m_n}{1-\beta_1^n} \\ \tilde{v}_n & = \dfrac{v_n}{1-\beta_2^n} \end{cases}$$

Also introduces **momentum** m_n, which is useful for unknown reasons.

Adam

Properties of Adam

Adam behaves just as Adagrad under scalar or diagonal reparametrization.

For a given number of iterations N, given $\beta_2 = (1 - 1/N)$ and $\gamma \propto 1/\sqrt{N}$, converges just as Adagrad $O\left(1/\sqrt{N}\right)$ without knowing L. [Defossez et al. 2020]

Same as **SGD with constant step size**: moves faster away from x_0 !

Adam

Properties of Adam

Intuitively: Adam moves each dimension by the **same amount** at every iteration.

Convergences requires $\beta_2 \to 1$. Default is 0.999 (average of 1000 samples).

Recently, $\beta_2 = 0.9$ became popular too: it is more important to move by a constant amount, than to converge.

Regularization of DNN

Generalization and stability

L2 regularisation

a.k.a weight decay

Historically: **Tikhonov regularization** for under determined least mean square regression, i.e. given $A \in \mathbb{R}^{n \times d}, y \in \mathbb{R}^n$:

$$\min_{x \in \mathbb{R}^d} \|Ax - y\|^2 + \lambda \|x\|^2.$$

Useful when $\mathrm{rank}(A) < d$.

More widely known as **L2 penalty** or **weight decay**. With SGD, equivalent to

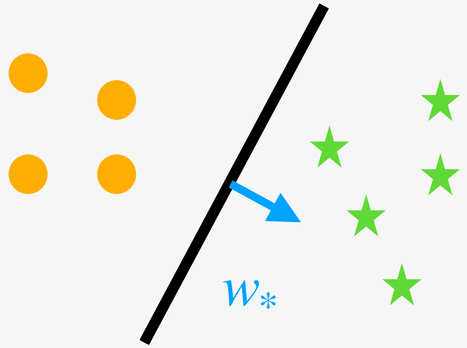
$$x_{n+1} = (1 - \gamma\lambda)x_n - \gamma \nabla f_n(x_n).$$

L2 regularisation

a.k.a weight decay

L2 regularisation has two roles:

- Improve stability (in previous example, allow to compute $(A^T A)^{-1}$),
- Improve generalization: Vapnik theory [Vapnik 1991], **constraints** on model class improve generalization.



Logistic regression for perfectly **separable** data points:

$$P(x = \bigstar) = \frac{\exp(x^T w_*)}{1 + \exp(x^T w_*)}.$$

Optimal solution verifies $\|w_*\| = \infty$

Regularization and DNN

Vapnik theory is not verified anymore (larger, more complex models can generalize better than smaller ones). Still generalization helps in some cases.

Sometime replaced by **early stopping** (keep best model on valid).

Extra stability issue: for a DNN, ∇F is never Liptchitz, because of **layer multiplications**.

Can lead to divergence even for Adam/Adagrad if change is too fast.

For Adam: bad interaction between denominator and L2 term, see AdamW [Loshchilov et Hutter, 2019].

Other regularizations

Related to Liptchitz factor

Spectral normalization: prevent eigenvalues in each layer to become too large (exactly controls overall Lipchitz factor) [Yoshida and Miyato, 2017].

WeightNorm: controls how quickly output scale can change [Salimans and Kingma, 2016].

$Y = WX$, with $X \in \mathbb{R}^d$, $W \in \mathbb{R}^{1\times d}$. Adam moves each each entry in W by γ, scale of Y moves by $d\gamma$ ($d \approx 1000$).
with **WeightNorm**: $\tilde{W} = S||W||^{-1}W$, with $S \in \mathbb{R}$, S moves at most by γ.

BatchNorm, **LayerNorm** etc: same + normalized scale for the output.

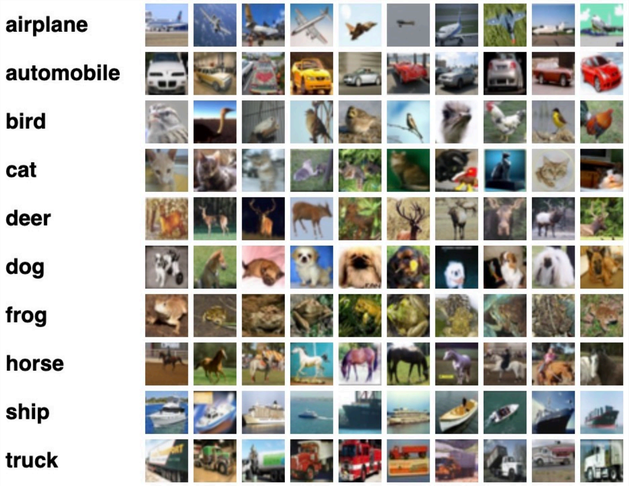
Practical DNN training

Getting the code

Go to github.com/adefossez/dnn_theo_practice to follow the code.

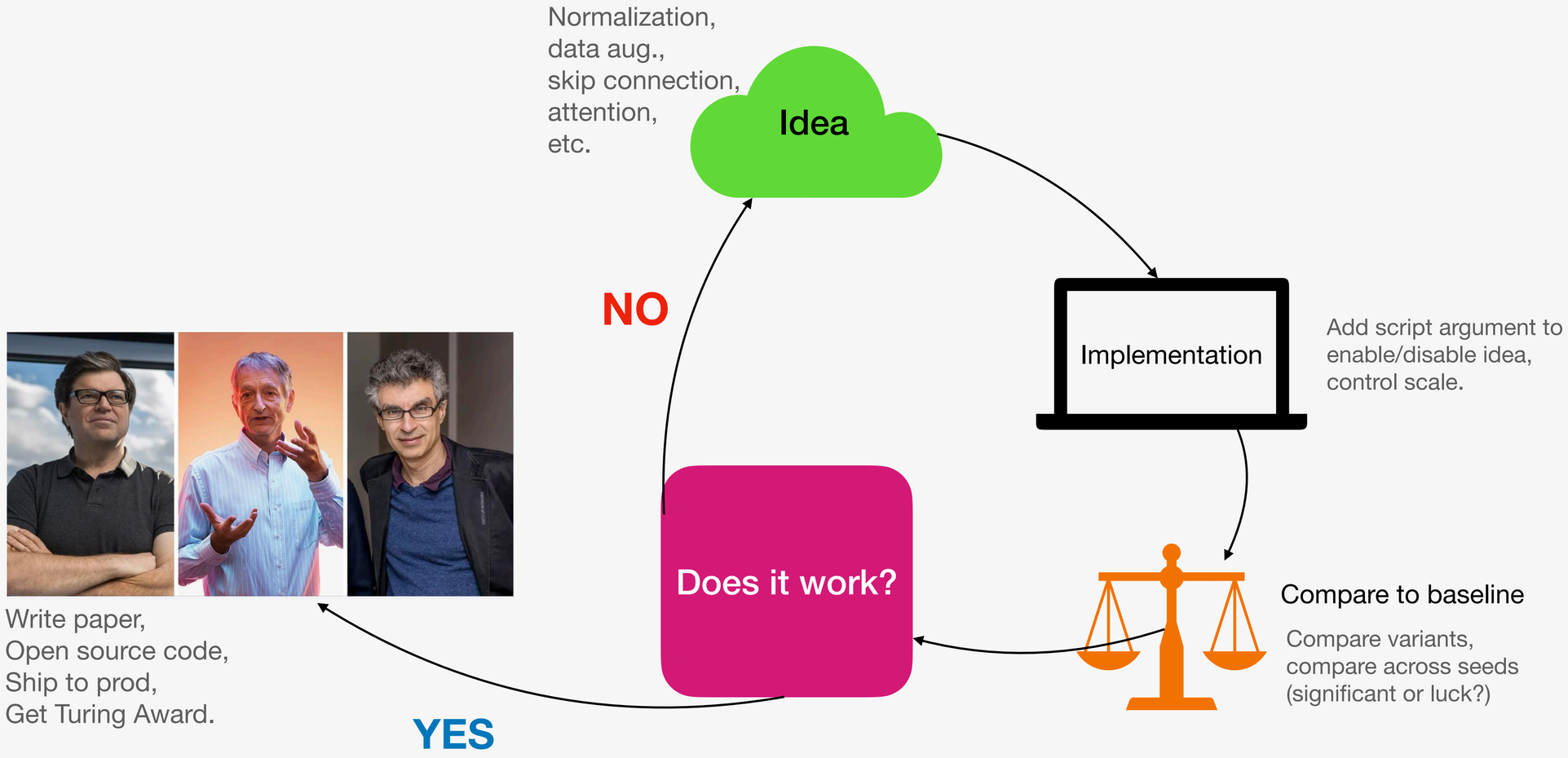
Provide a basic training loop using no framework.

Also an example using PyTorch-Lightning and Hydra.

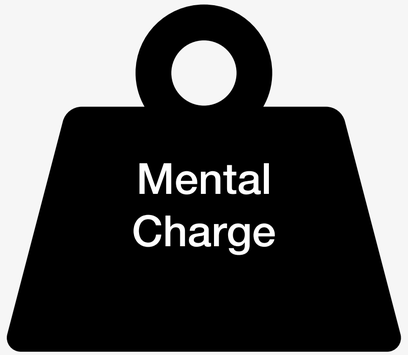



```python
def do_epoch(epoch, model, loader, optimizer=None):
    """Run a single epoch, either in training or evaluation mode, if `optimizer` is None."""

    device = next(model.parameters()).device

    average = averager()

    for input_, label in loader:
        input_ = input_.to(device)
        label = label.to(device)

        prediction = model(input_)

        loss = F.cross_entropy(prediction, label)
        predicted_label = prediction.argmax(dim=1)
        accuracy = (label == predicted_label).float().mean()

        metrics = {
            'loss': loss,
            'accuracy': accuracy,
        }
        metrics = average(metrics)

        if optimizer is not None:
            loss.backward()
            optimizer.step()
            optimizer.zero_grad()

    label = 'test' if optimizer is None else 'train'
    print(f'[{label: <5}] {epoch:04d}, '
          f'loss: {metrics["loss"]:.3f}, '
          f'acc.: {metrics["accuracy"]:6.2%}')
    return metrics
```

The Experimental Research Process



Pitfalls of experimental research



- 1 out 10 ideas works.
- Can take many cycles to work.
- Task variations (different datasets, models, etc).
- Experiment duration range from a few hours to several weeks.

What do we need to succeed?

- Easily try many variants and combinations (a.k.a. grid search).
- Exploit parallelism of a cluster.
- Easily keep track of experiments, compare and plot.

  ➡ Draw conclusion on what to try next.
- Resume interrupted experiments.

Improved training loop

- Each experiment should have a **name** (automatic ideally).
- Store **logs** and **checkpoints** using this name.
- Later, we can figure out which logs comes from which XP.
- Can **resume** interrupted training (error, crash, preemption).

Grid searches

- Grid search: cartesian product of hyper-parameters.
- With a cluster, you can test many experiments in parallel.
- **Be smart:** choose 1 or 2 hyper-parameters at once, then freeze them and continue (develop and use intuition).

Distributed

- Experiments can quickly take several weeks.
- Distributed over G gpus: given batch B, split it in G groups of B / G.
- Compute gradient in parallel, average and sync gradient.
- Distributed Data Parallel: each GPU has its own process. All processes run the same code.
- On single machine, simpler to use Data Parallel.

Tooling: Hydra

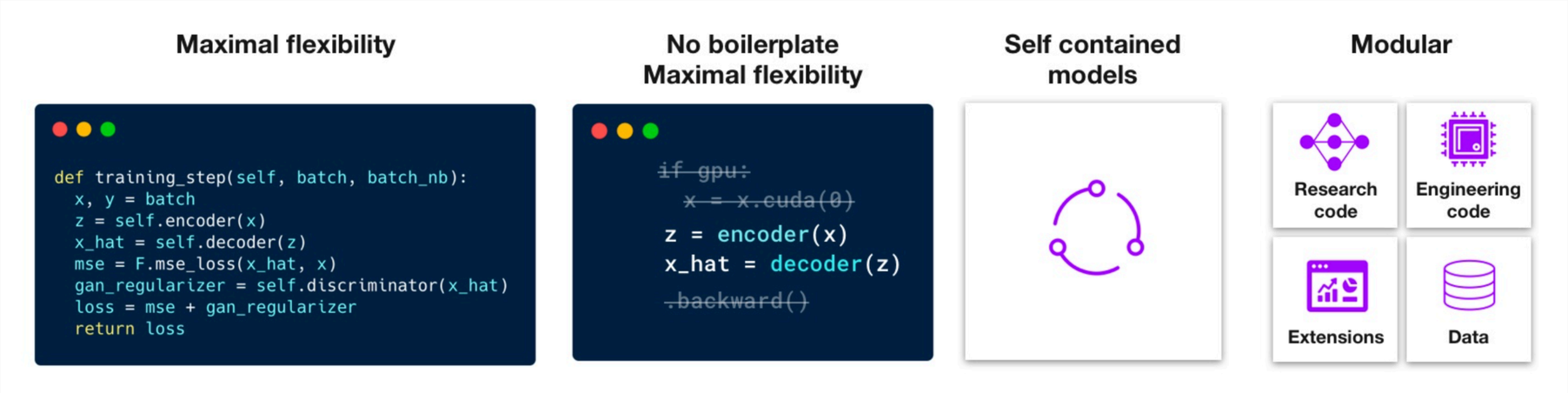
https://github.com/facebookresearch/hydra

- Hydra provide hierachical YAML based configuration (YAML is nicer than JSON for humans).
- Also provide logging, basic grid search support from the command line.
- Integrates with meta-optimizers like Nevergrad.

Tooling: PyTorch-Lightning

https://github.com/PyTorchLightning/pytorch-lightning

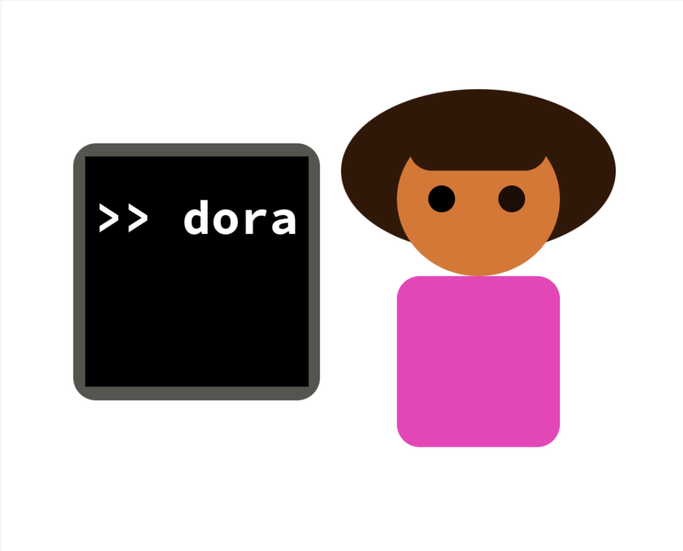
- Remove all boilerplate for checkpoints, logging, distributed etc.
- But you will lose flexibility and understanding.



Tooling: Dora

https://github.com/facebookresearch/dora/

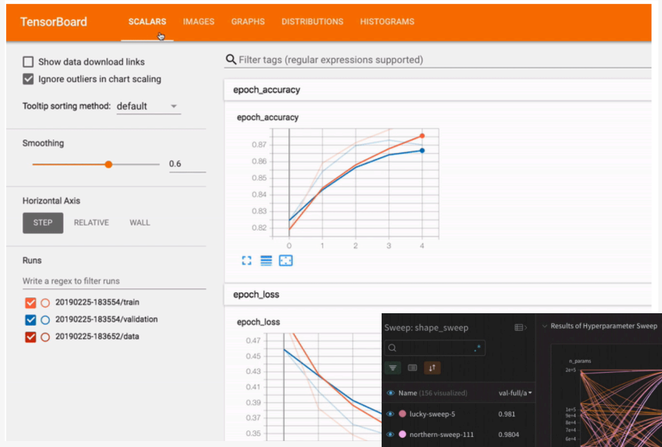
- Defines grid search as python files.
- XP identified by unique **signature** hash.
- What runs on the cluster is what you want.
- Basic reporting from the terminal.

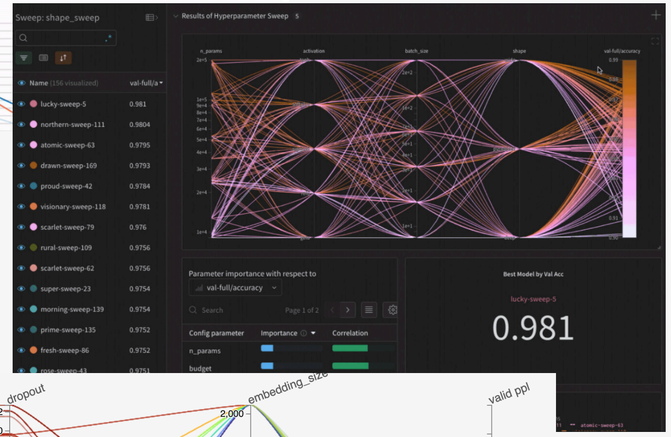


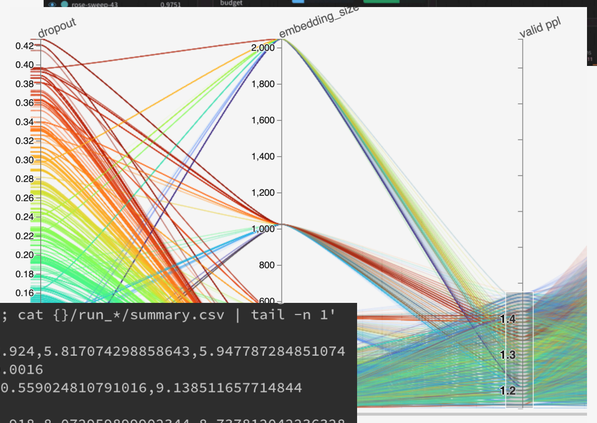

```
(env) → @dev[intro_practical_dl]/intro_practical_dl git:(dora) dora grid my_grid
Monitoring Grid my_grid
                                Meta                                |     |     train      |      test
i  name                                  sta       sig       sid | epo |  loss  accurac |  loss  accura
0                                        COM  97d170e1  46014124 | 100 | 0.273   90.75% | 0.890  74.09%
1  batch_size=64                         COM  74a74e3c  46014125 | 100 | 0.065   97.91% | 1.186  73.11%
2  lr=0.01                               COM  f5b313bd  46014126 | 100 | 0.001  100.00% | 1.690  64.76%
3  batch_size=64 lr=0.01                 COM  4772033a  46014127 | 100 | 0.001  100.00% | 2.055  59.70%
4  model=mobilenet_v2                    COM  9ead75da  46010996 | 100 | 0.494   83.18% | 0.855  72.97%
5  batch_size=64 model=mobilenet_v2      COM  9201e92e  46011004 | 100 | 0.259   91.03% | 0.944  73.95%
6  lr=0.01 model=mobilenet_v2            COM  010000f3  46014128 | 100 | 0.030   99.24% | 2.238  60.96%
7  batch_size=64 lr=0.01 model=mobilenet_v2 COM  0d303554  46011006 | 100 | 0.001  100.00% | 2.912  56.65%
```

Tooling: Visualization

- Tensorboard: initially for tensorflow, also for PyTorch and PyTorch-Lightning.
- Wandb: track experiment in your browser.
- HiPlot: compelling way of making sense of the impact of hyper params on model performance.
- Good old command line tools. Don't neglect those (grep, tmux, bashrc etc).
- See fd, ag or rg.








```
@dev/timm2 fd -d 1 '.*' -x sh -c 'echo {}; cat {}/run_*/summary.csv | tail -n 1'
xp_penalty=0.1_group-size=8_pretrained
5,3.5048252003533498,1.0108134375,75.292,92.924,5.817074298858643,5.947787284851074
xp_penalty=0.1_group-size=8_pretrained_lr=0.0016
,3.4995868035725186,0.89917,81.158,95.658,10.559024810791016,9.138511657714844
xp_penalty=0.01_group-size=8
7,2.8587126902171542,0.9513128125,77.382,93.918,8.072959899902344,8.737812042236328
xp_penalty=0.5_group-size=8
7,4.970405987330845,1.2356575,70.992,90.544,3.727231979370117,3.8965940475463867
xp_penalty=0.05_group-size=8
9,3.0909746885299683,1.01999625,76.374,93.356,5.795430660247803,6.120972633361816
xp_penalty=0.1_group-size=8
7,3.307918225015913,1.1356646875,75.26,92.75,4.980111598968506,5.214414596557617
@dev/timm2
```

“Einsum”: tensor product super powers

- Self descriptive, generic (**outer** and **inner** product).
- One example for coding **attention** yourself:

```
import torch

def attention(queries, keys, values):
    # String describes operation to perform using Einstein notation.
    # bct is first input shape [B, C, T]. Then second input.
    # After ->, output. c disappears, so inner product on c.
    # For keys, we use a second name for time `s`, and keep
    # `ts` in  output: outer product on time steps.
    scores = torch.einsum("bct,bcs->bts", queries, keys)
    scores = torch.softmax(scores, dim=2)
    result = torch.einsum("bts,bcs->bct", scores, values)

queries = torch.randn(32, 64, 344)
keys = torch.randn_like(queries)
values = torch.randn_like(keys)
attention(queries, keys, values)
```

That’s it!

Code and slides: github.com/adefossez/dnn_theo_practice